UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

FABRÍCIO NEGRISOLO DE GODÓI

ESTUDO COMPARATIVO ENTRE CLUSTERS DE COMPUTADORES DESKTOP E DE DISPOSITIVOS ARM

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2015

FABRÍCIO NEGRISOLO DE GODÓI

ESTUDO COMPARATIVO ENTRE CLUSTERS DE COMPUTADORES DESKTOP E DE DISPOSITIVOS ARM

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Diplomação 2, do Curso Superior de Engenharia de Computação, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de bacharel.

Orientador: Prof. Me. Adriano Serckumecka
Coorientador: Prof. Dr. Marco Antônio de Castro Barbosa

PATO BRANCO
2015

MINISTÉRIO DA EDUCAÇÃO
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

TERMO DE APROVAÇÃO

Às 9 horas do dia 26 de junho 2015, na sala V108, da Universidade Tecnológica Federal dão Paraná, Câmpus Pato Branco,. reuniu-se a banca examinadora composta pelos professores Adriano Serckumecka (orientador), Marco Antonio de Castro Barbosa (coorientador), Éden Ricardo Dosciatti e Fábio Favarim para avaliar o trabalho de conclusão de curso com o título **Estudo comparativo entre clusters de computadores desktop e de dispositivos ARM**, do aluno **Fabrício Negrisolo de Godoi**, matrícula 1209434, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

Adriano Serckumecka
Orientador (UTFPR)

Marco Antonio de Castro Barbosa
Coorientador (UTFPR)

Éden Ricardo Dosciatti
(UTFPR)

Fábio Favarim
(UTFPR)

Beatriz Terezinha Borsoi
Coordenador de TCC

Marco Antonio de Castro Barbosa
Coordenador do Curso de
Engenharia de Computação

AGRADECIMENTOS

Primeiramente aos meus pais e irmãos, Alisson, Fábio e Felipe, que me apoiaram e proporcionaram condições de realizar este curso.

À minha namorada e companheira Geórgia A. C. Zangaro, por me ajudar e animar nas horas mais difíceis.

Aos meus professores Adriano Serckumecka, Marco A. C. Barbosa, Beatriz T. Borsoi e Diogo R. Vargas, que me auxiliaram durante a realização deste trabalho.

À esta universidade, seu corpo docente, direção e administração que me guiaram, ensinaram e proporcionaram todas as ferramentas necessárias para minha graduação.

Quanto mais aumenta nosso conhecimento, mais evidente fica nossa ignorância.

John F. Kennedy

RESUMO

GODÓI, Fabrício Negrisolo de. Estudo comparativo entre *clusters* de computadores *desktop* e de dispositivos ARM. 2015. 117 f. Monografia (Trabalho de Conclusão de Curso) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2015.

Este trabalho apresenta uma comparação de custo/benefício entre um *cluster* de computadores *desktop* tradicionais (computadores HP Compaq 6005 Pro Microtower) e de dispositivos de baixo consumo de energia, como Raspberry Pi Modelo B e Cubietruck. Para testá-los, foram implementados algoritmos para resolução de problemas das classes P e NP-Difícil. Foram realizadas análises com os dados obtidos como, por exemplo, o cálculo de *speedup*. Com os dados coletados de ambos os *clusters* foi possível estimar o custo e benefício gerado, bem como verificar sua aplicabilidade.

Palavras-chave: Cluster. Dispositivos ARM. Energia. MPI.

ABSTRACT

GODÓI, Fabrício Negrisolo de. Comparative study between desktop computers and ARM devices clusters. 2015. 117 f. Monografia (Trabalho de Conclusão de Curso) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2015.

This work presents a comparison of cost/benefit ratio between a traditional desktop computers (HP Compaq 6005 Pro Microtower computer) cluster and a low-power consumption devices cluster, like the Raspberry Pi Model B and Cubietruck. In order to test them, algorithms to solve P and NP-Hard classes of problems have been implemented. Analyzes were performed with the obtained data as, for example, the speedup calculation. With the collected data from both clusters was possible to estimate the cost and benefit generated and their applicability.

Keywords: ARM Devices. Cluster. Energy. MPI.

LISTA DE FIGURAS

FIGURA 1 – MODELO DE CIRCUITO BOOLEANO 24
FIGURA 2 – REPRESENTAÇÃO DO CAIXEIRO VIAJANTE 25
FIGURA 3 – REPRESENTAÇÃO DOS *CLUSTERS* 35
FIGURA 4 – ESQUEMA LÓGICO PARA CAPTURA DE ENERGIA. 36
FIGURA 5 – DESCRIÇÃO DA REDE DE COMUNICAÇÃO ENTRE DISPOSITIVOS. 39
FIGURA 6 – TELA PUTTY 40
FIGURA 7 – TERMINAL DE COMUNICAÇÃO DO PUTTY 40
FIGURA 8 – TELA WINSCP 41
FIGURA 9 – TELAS DE COMUNICAÇÃO, EXECUÇÃO E CAPTURA DE DADOS. 43
FIGURA 10 – TERMINAL JAVA. 43
FIGURA 11 – FLUXOGRAMA MICROCONTROLADOR 53
FIGURA 12 – DIAGRAMA DE CLASSES JCENTRAL 54
FIGURA 13 – FLUXOGRAMA JCENTRAL 55
FIGURA 14 – GRÁFICO DO CUSTO EM R$ DE CADA TIPO DE *CLUSTER* 59
FIGURA 15 – GRÁFICO, TODOS OS RESULTADOS HP LINPACK 61
FIGURA 16 – GRÁFICO, RESULTADOS HP LINPACK RASPBERRY E CUBIETRUCK 62
FIGURA 17 – GRÁFICOS DE TEMPO PARA O PROBLEMA DE MULTIPLICAÇÃO DE MATRIZ 65
FIGURA 18 – GRÁFICOS DE *SPEEDUP* PARA O PROBLEMA DE MULTIPLICAÇÃO DE MATRIZ 67
FIGURA 19 – GRÁFICOS DE TEMPO PARA O PROBLEMA DO CAIXEIRO VIAJANTE 69
FIGURA 20 – GRÁFICOS DE *SPEEDUP* PARA O PROBLEMA DO CAIXEIRO VIAJANTE 71
FIGURA 21 – RESULTADOS CUSTO ENERGÉTICO PARA MULTIPLICAÇÃO DE MATRIZ 75
FIGURA 22 – GRÁFICOS, RESULTADOS CUSTO ENERGÉTICO PARA CAIXEIRO VIAJANTE 78

LISTA DE QUADROS

QUADRO 1 – ESPECIFICAÇÕES COMPUTADORES *DESKTOP* HP.....31
QUADRO 2 – ESPECIFICAÇÕES RASPBERRY PI, MODELO B.....32
QUADRO 3 – ESPECIFICAÇÕES CUBIETRUCK.....32
QUADRO 4 – SISTEMAS E COMPONENTES UTILIZADOS.....32
QUADRO 5 – CORAÇÃO DOS EQUIPAMENTOS.....33
QUADRO 6 – FERRAMENTAS UTILIZADAS.....34
QUADRO 7 – COMO EXECUTAR OS PROGRAMAS.....41
QUADRO 8 – MULTIPLICAÇÃO DE MATRIZ SEQUENCIAL.....45
QUADRO 9 – MULTIPLICAÇÃO DE MATRIZ PARALELA.....46
QUADRO 10 – CAIXEIRO VIAJANTE SEQUENCIAL.....48
QUADRO 11 – CAIXEIRO VIAJANTE PARALELO.....51
QUADRO 12 – CONFIGURAÇÃO DO MICROCONTROLADOR.....52
QUADRO 13 – CONFIGURAÇÃO ARQUIVO /ETC/HOSTS.....56
QUADRO 14 – CONFIGURAÇÃO /ETC/NETWORK/INTERFACES.....56

LISTA DE TABELAS

TABELA 1 – CUSTO EM R$ DE CADA *CLUSTER*.....33
TABELA 2 – CUSTO EM R$ DE CADA TIPO DE *CLUSTER* 59
TABELA 3 – RESULTADOS HP LINPACK DO *CLUSTER DESKTOP* 60
TABELA 4 – RESULTADOS HP LINPACK DO *CLUSTER* RASPBERRY 60
TABELA 5 – RESULTADOS HP LINPACK DO *CLUSTER* CUBIETRUCK.....61
TABELA 6 – TEMPO EM SEGUNDOS (*S*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER DESKTOP* 64
TABELA 7 – TEMPO EM SEGUNDOS (*S*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* RASPBERRY 64
TABELA 8 – TEMPO EM SEGUNDOS (*S*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* CUBIETRUCK 64
TABELA 9 – *SPEEDUP* DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER DESKTOP*.....66
TABELA 10 – *SPEEDUP* DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* RASPBERRY.....66
TABELA 11 – *SPEEDUP* DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* CUBIETRUCK.....66
TABELA 12 – TEMPO EM SEGUNDOS (*S*) DO CAIXEIRO VIAJANTE NO *CLUSTER DESKTOP* 68
TABELA 13 – TEMPO EM SEGUNDOS (*S*) DO CAIXEIRO VIAJANTE NO *CLUSTER* RASPBERRY 68
TABELA 14 – TEMPO EM SEGUNDOS (*S*) DO CAIXEIRO VIAJANTE NO *CLUSTER* CUBIETRUCK 68
TABELA 15 – *SPEEDUP* DO CAIXEIRO VIAJANTE NO *CLUSTER DESKTOP*.....70
TABELA 16 – *SPEEDUP* DO CAIXEIRO VIAJANTE NO *CLUSTER* RASPBERRY .70
TABELA 17 – *SPEEDUP* DO CAIXEIRO VIAJANTE NO *CLUSTER* CUBIETRUCK 70
TABELA 18 – POTÊNCIA MÉDIA EM *WATTS* DOS *CLUSTERS* EM REPOUSO E TRABALHANDO 73
TABELA 19 – CUSTO ENERGÉTICO (*WS*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER DESKTOP* 74
TABELA 20 – CUSTO ENERGÉTICO (*WS*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* RASPBERRY 74
TABELA 21 – CUSTO ENERGÉTICO (*WS*) DA MULTIPLICAÇÃO DE MATRIZ NO *CLUSTER* CUBIETRUCK 74
TABELA 22 – CUSTO ENERGÉTICO (*WS*) DO CAIXEIRO VIAJANTE NO *CLUSTER DESKTOP* 77
TABELA 23 – CUSTO ENERGÉTICO (*WS*) DO CAIXEIRO VIAJANTE NO *CLUSTER* RASPBERRY 77
TABELA 24 – CUSTO ENERGÉTICO (*WS*) DO CAIXEIRO VIAJANTE NO *CLUSTER* CUBIETRUCK 77

LISTA DE SIGLAS, ABREVIATURAS E ACRÔNIMOS

ADC	*Analog to Digital Converter*
ARM	*Advanced RISC Machine*
BLAS	*Basic Linear Algebra Subprograms*
FLOPS	*FLoting-point Operation Per Second*
GRASP	*Greedy Randomized Adaptive Search Procedure*
GFLOPS	*Billions FLoting-point Operation Per Second*
MFLOPS	*Millions FLoting-point Operation Per Second*
MIMD	*Multiple Instruction stream Multiple Data stream*
MIPS	*Millions of Instructions Per Second*
MISD	*Multiple Instruction stream Multiple Data stream*
MPI	*Message Passing Interface*
NOW	*Network of Workstations*
NP	Tempo polinomial não determinístico
OpenMP	*Open Multi-Processing*
PCV	Problema do Caixeiro Viajante
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
RMS	*Root Mean Square*
ROM	*Read-only memory*
SIMD	*Single Instruction stream Multiple Data stream*
SISD	*Single Instruction stream Single Data stream*
SoC	*System-on-a-chip*
SSH	*Secure Shell*
TSP	*Traveling Salesman Problem*
UART	*Universal Assynchronous Receiver/Transmitter*
ULA	Unidade Lógica Aritmética
UPF	Unidade de Ponto Flutuante
VLIW	*Very Long Instruction Word*
VSIPL	*Vector Signal Image Processing Library*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 CONSIDERAÇÕES INICIAIS 13
1.2 OBJETIVOS 15
1.2.1 Objetivo Geral 15
1.2.2 Objetivos Específicos 15
1.3 JUSTIFICATIVA 16
1.4 ESTRUTURA DO TRABALHO 17
2 REFERENCIAL TEÓRICO 18
2.1 PARALELISMO 18
2.2 CLUSTERIZAÇÃO 20
2.3 COMPLEXIDADE DE ALGORITMOS E META-HEURÍSTICAS 22
2.4 COMPLEXIDADE DE ALGORITMOS PARALELOS 23
2.5 O PROBLEMA DO CAIXEIRO VIAJANTE 24
2.6 SPEEDUP 25
2.7 HP LINPACK 26
2.8 MICROCONTROLADORES 26
2.8.1 Conversor Analógico/Digital 26
2.8.2 Transmissor/Receptor Assíncrono Universal 28
2.9 EFEITO HALL 28
2.10 VALOR EFICAZ 29
3 MATERIAIS E MÉTODOS 31
3.1 MATERIAIS 31
3.2 MÉTODOS 34
4 DESENVOLVIMENTO 38
4.1 DESCRIÇÃO DO SISTEMA 38
4.2 APRESENTAÇÃO 39
4.3 IMPLEMENTAÇÃO 44
4.3.1 Multiplicação de Matriz 44
4.3.2 Caixeiro Viajante 46
4.3.3 Sistema de Captura de Dados 52
4.3.4 *Clusters* 55
5 DADOS OBTIDOS E DISCUSSÕES 58
5.1 CUSTOS DE CONSTRUÇÃO DOS CLUSTERS 58
5.2 RESULTADOS HP LINPACK 60
5.3 RESULTADOS DE TEMPO, SPEEDUP E ENERGIA 63
5.3.1 Resultados de tempo e *speedup* 64
5.3.2 Resultados custo energético 72
6 CONCLUSÃO 80
REFERÊNCIAS 82
APÊNDICES 87
APÊNDICE A – Código do microcontrolador Tiva 88
ANEXOS 93
ANEXO A – Exemplo de entrada para Multiplicação de Matriz 94
ANEXO B – Exemplo de entrada para o problema do Caixeiro Viajante 94

1 INTRODUÇÃO

Neste capítulo são apresentadas as considerações iniciais do trabalho, os objetivos, suas justificativas e, ao final, a estrutura geral do trabalho.

1.1 CONSIDERAÇÕES INICIAIS

Para resolver problemas complexos, sendo eles do cotidiano, empresariais, ou mesmo acadêmicos, máquinas computacionais cada vez melhores, alto desempenho e baixo custo, são requisitadas (SLOAN, 2004). Apesar dos processadores terem sido aprimorados em suas fontes de *clock,* tornando-os mais rápidos, eles possuem limitações físicas provenientes dos circuitos eletrônicos utilizados. Por esses motivos é inviável aumentar o nível de *clock* indefinidamente, pois o processador superaquece e consome maior quantidade de energia, sendo necessário resfriamento monitorado para evitar alguns possíveis danos ao processador (RAUBER; RÜNGER, 2010, p. 22; INTEL, 2014).

Um método para contornar este problema é a paralelização de algoritmos, ou seja, dividir um código em pedaços menores e distribuí-lo a vários processadores ou máquinas (RAUBER; RÜNGER, 2010). Para isto, sistemas distribuídos (*cluster*), utilizando computadores de propósito geral (*desktop*) são alternativas praticáveis para atingir o alto desempenho. Entretanto, esta medida tem um custo energético elevado, tanto para alimentá-las, quanto para resfriá-las e isto, a longo prazo, compromete seu custo/benefício. O ramo de eficiência energética vem se destacando ao longo dos anos buscando alternativas para reduzir o consumo de energia, pois o consumo tende a extrapolar a produção (PANDE et al., 2011; WANG; FENG; XUE, 2011).

Visando contornar estes problemas, foram desenvolvidos dispositivos embarcados a frio, como Raspberry, BeagleBone e Cubieboard, que não necessitam de resfriadores e dissipadores de calor de grande potência, além de consumirem uma quantia de energia menor que os *desktop* (RASPBERRY PI, 2014; TEXAS INSTRUMENTS, 2014; CUBIEBOARD, 2014). Unindo as qualidades desses

dispositivos, com as qualidades que um sistema paralelo e distribuído oferecem, pode-se alcançar considerável poder de processamento. Apesar deste poder ser inferior se comparados ao mesmo número de computadores de propósito geral, suas características podem auxiliar na redução de consumo de energia elétrica, reduzindo custos e consumos de forma geral. Outro fator a ser levado em conta, é o espaço físico reduzido desses dispositivos em relação aos computadores *desktop*. Enquanto um Raspberry Pi possui dimensões de 85mm x 56mm x 21mm, ou mesmo uma placa BeagleBone Black possui 86,36mm x 53,34mm, o computador *desktop* Aspire MC605 possui 380,5mm x 175mm x 413,7mm (RASPBERRY PI, 2014; TEXAS INSTRUMENTS, 2014; ACER, 2014).

Muitos dos problemas que necessitam desse poder de processamento elevado são encontrados no cotidiano, tais como: de logística em empresas, carregamento e entregas, roteamento de veículos de socorro, cobertura e posicionamento de radares, dentre outros relativamente comuns. Esses problemas constituem-se no maior desafio da Teoria da Computação e podem ser relacionados com problemas computacionais clássicos, tais como: o Problema do Caixeiro Viajante (para otimização de rotas de aviões, transportadoras, etc.), Circuito Hamiltoniano (cálculo da menor trajetória pelo GPS), Problema da Cobertura de Vértices (posicionamento de câmeras, radares, antenas, etc.), entre outros (GAREY; JOHNSON, 1979). Problemas desta natureza podem ser categorizados como NP-Completos ou NP-Difíceis, e embora sejam considerados intratáveis por demandarem tempo de execução exponencial, suas soluções são necessárias no dia a dia das pessoas e, portanto, requerem respostas viáveis em tempos aceitáveis.

Existem alternativas para se buscar soluções para estes problemas, sendo elas:

a) Algoritmos de solução exata: nesta abordagem a solução obtida será sempre a ótima, mas o tempo necessário para se obter tal resposta ainda é exponencial. São exemplos desta técnica os algoritmos: *backtracking*, *branch-and-bound* e *branch-and-cut*, podendo ser encontradas em Cormen (2002).
b) Heurística e meta-heurísticas: são técnicas computacionais que fornecem uma solução viável, não necessariamente a ótima, porém o tempo é aceitável (polinomial). Exemplos destas técnicas são as meta-heurísticas: GRASP (*Greedy Randomized Adaptive Search Procedure*), Algoritmos Genéticos, Colônia de Formigas, *Simulated Annealing*, dentre outras, como pode ser encontrado em Blum e Roli (2003) e Glover e Kochenberger (2002).

c) Paralelismo: as técnicas de paralelismo podem ser empregadas tanto para algoritmos exatos (BADER; HART; PHILLIPS, 2005), quanto para métodos heurísticos (AIEX; BINATO; RESENDE, 2003), dependendo do tipo de aplicação a ser usada. Em uma arquitetura paralela, é possível utilizar métodos exatos para encontrar soluções para problemas complexos, entretanto isso só é possível para um conjunto restrito de dados do problema, que está relacionado diretamente com a quantidade de processadores disponíveis.

Com base nessas premissas, o objetivo deste trabalho, assim como nos trabalhos de Padoin et al. (2012), Ou et al. (2012) e Cox et al. (2013), é a construção de um *cluster* de alto desempenho, visando duas principais características: desempenho e custo energético.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral desse trabalho é a avaliação da relação custo/benefício entre um *cluster* formado por dispositivos ARM versus um *cluster* de computadores *desktop*, visando duas principais características: desempenho computacional e custo energético.

1.2.2 Objetivos Específicos

Preparação do *hardware* envolvido:

- Projetar e desenvolver um dispositivo para medir a energia gasta nos equipamentos utilizados neste trabalho;
- Instalar e configurar as máquinas envolvidas no *cluster desktop*, para realização dos testes sequenciais e paralelizados;

- Instalar e configurar os dispositivos ARM em *cluster*, para realização dos testes sequenciais e paralelos.

Objetos de testes:

- Codificar soluções para problemas da classe P e NP-Difícil (em linguagem C) para execução sequencial e paralela;
- Utilizar o *benchmark* HP Linpack nos *clusters desktop* e ARM para obter parâmetros de desempenho.

Realização dos testes e coleta de dados:

- Avaliar o desempenho das arquiteturas em *cluster* com a execução dos algoritmos codificados, através da coleta dos dados: tempo de execução e custo energético;
- Relacionar a cada arquitetura o custo financeiro do *hardware* envolvido.

1.3 JUSTIFICATIVA

Para resolver problemas complexos, sendo eles do cotidiano, empresariais, ou mesmo acadêmicos, máquinas computacionais cada vez melhores (alto desempenho e baixo custo) são utilizadas para estes propósitos. De acordo com Sloan (2004), a necessidade de realizar cálculos rápidos não é só um luxo, também é uma corrida, tanto empresarial, quanto acadêmica, que pode decidir quem publicará primeiramente um artigo, uma patente, etc. Como máquinas possuem limites físicos que as impedem de atingirem certos níveis de processamento, surgiram os *clusters* computacionais que são compostos por máquinas que cooperam para resolver um mesmo problema. Com os *clusters* é possível realizar cálculos complexos e demorados num tempo significativamente menor, entretanto é necessária uma quantia significativa de energia para mantê-los, tanto para funcionar quanto para resfriá-los.

Como alternativa para os computadores tradicionais que necessitam de constante resfriamento, são estudados e fabricados sistemas embarcados de baixo consumo de energia, que por não utilizarem quantias significativas de energia,

acabam gerando menor quantidade de calor. Padoin et al. (2012) descreve uma comparação entre PandaBoard© e BeagleBoard© demonstrando o baixo consumo energético de ambas plataformas. Apesar de terem um desempenho relativamente menor, seu custo, tamanho, peso, energia gasta, são ainda menores, fazendo com que seja viável a compra desses equipamentos em relação aos *desktops*. Realizando uma clusterização com esses sistemas embarcados, é possível alcançar níveis de processamento equivalentes a uma clusterização convencional, entretanto com a necessidade de uma quantidade maior de dispositivos.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em cinco capítulos, sendo que o segundo e o terceiro capítulos realizam uma breve introdução às teorias e aos materiais e métodos utilizados. E por fim, são realizadas as coletas de dados e suas análises, assim como o resultado proveniente da pesquisa nos capítulos quatro e cinco.

2 REFERENCIAL TEÓRICO

Neste capítulo consta o referencial teórico dos seguintes assuntos: paralelismo, clusterização, complexidade de algoritmos e meta-heurísticas e, por fim, complexidade de algoritmos paralelos.

2.1 PARALELISMO

Na matemática, o termo paralelo é geralmente utilizado para descrever quando duas retas nunca se intersectam, na computação o termo é utilizado para descrever quando mais de uma operação ocorre ao mesmo tempo. A paralelização pode ocorrer tanto em *hardware* quanto em *software*, sendo que o primeiro caso não é controlável pelo usuário e o segundo é realizado no próprio desenvolvimento do código pelo usuário. De acordo com Rauber et al. (2010), há quatro níveis em que a paralelização pode ocorrer em hardware, sendo elas:

1) Paralelismo a nível de bit: Referente ao tamanho da palavra utilizada pelo processador para realizar as operações que receberam incrementos significativos, desde 4 bits até 64 bits (atualmente). Houve esse aumento por causa da necessidade de endereçar cada vez mais instruções em uma mesma linha de código, e também para que fosse possível adquirir maior precisão de operações que utilizam ponto flutuante.
2) Paralelismo por *pipeline*: O *pipeline* é realizado pelo próprio *hardware*, fazendo com que diversas instruções de um código ocorram ao mesmo tempo em diferentes dispositivos dedicados (estágios de *pipeline*). Cada instrução é “quebrada” em partes menores, para que seja possível que cada dispositivo realize seu processamento de forma independente. O tempo do *pipeline* é o que define o tempo de ciclo do processador, pois cada estágio do *pipeline* deve utilizar o mesmo tempo de processamento. O grau de paralelismo é definido pelo número de estágios utilizado no *pipeline*, que muitas vezes são chamados de *superpipelined*, por causa da quantidade de estágios que possuem. Apesar do grau de paralelismo crescer consideravelmente com o

aumento do número de estágios, isso possui um limite por causa das dependências de instruções.

3) Paralelismo por múltiplas unidades funcionais: As unidades funcionais podem ser classificadas como: ULAs (Unidades Lógicas Aritméticas), UPFs (Unidades de Ponto Flutuante), unidades de carregamento e armazenamento, etc. Utilizando diversas unidades desse tipo é possível realizar paralelização das instruções de maneira independente. Esse tipo de paralelismo pode ser classificado em: superescalar e VLIW (V*ery Long Instruction Word*). Assim como no *pipeline,* há uma restrição com o aumento da quantidade de unidades a serem adicionadas, nesse caso havendo dependências entre os dados das unidades vizinhas.
4) Paralelismo a nível de processo ou *thread*: Outra alternativa para realização da paralelização é aumentar o número de processadores, ou mesmo os núcleos de uma máquina, de forma a serem totalmente independentes uns dos outros. Máquinas com múltiplos processadores são utilizadas há alguns anos em servidores, entretanto atualmente já é comum aparelhos eletrônicos para fins pessoais possuírem processadores com diversos núcleos de processamento. Enquanto os níveis de paralelismo citados são independentes do usuário em questão, neste nível é necessário que o usuário utilize técnicas de paralelização de códigos para que processos sejam distribuídos entre os processadores, ou núcleos. Apesar dos processadores serem independentes entre si, ainda há a memória que é compartilhada por ambos, gerando uma necessidade de sincronização entre os mesmos para realizar o acesso à memória.

Flynn (1972) classifica de forma bem ampla os tipos de arquiteturas utilizadas, sendo elas:

a) SISD (*Single Instruction stream Single Data stream*): Representa os computadores mais tradicionais, em que é trabalhado um processo por vez e o nível de paralelismo se estende até o terceiro nível, como citado anteriormente.
b) SIMD (*Single Instruction stream Multiple Data stream*): Também definidas como máquinas vetoriais, foram amplamente usadas e definidas como supercomputadores. Neste caso, múltiplos dados são processados de uma

única vez, mas utilizando somente um tipo de operação. Amplamente utilizado para cálculos matemáticos e científicos.

c) MISD (*Multiple Instruction stream Single Data stream*): São máquinas que necessitam realizar diversas operações sobre o mesmo dado. Mesmo sendo difícil achar um exemplo devidamente aplicado na prática, pode-se associar como uma máquina para tratamento a falhas, com as instruções sendo executadas redundantemente.
d) MIMD (*Multiple Instruction stream Multiple Data stream*): São múltiplas operações realizadas em múltiplos dados, ou seja, as operações podem ocorrer de forma independente umas das outras. Computadores com múltiplos processadores ou mesmo processadores com múltiplos núcleos fazem parte desta classificação.

Dadas as classificações para os tipos de arquiteturas paralelas, resta saber como medir o desempenho de cada uma. Para muitos especialistas a melhor forma de medir o desempenho de uma máquina paralela é calcular a quantidade de instruções executadas dentro de um intervalo de tempo especifico. As duas unidades de medidas mais conhecidas e utilizadas são: MIPS (milhões de instruções executadas num segundo) e MFLOPS (milhões de instruções de ponto flutuante executadas num segundo), sendo que não existe uma relação exata entre uma e outra. Como as instruções variam de uma máquina para outra, é necessário utilizar programas padrão para medir o desempenho e compará-las. Essa medida é conhecida como *Benchmarking*. Os principais programas de *Benchmarks* utilizados atualmente são: *LINPACK, NAS e SPEC*, etc. (ROSE; NAVAUX, 2008).

2.2 CLUSTERIZAÇÃO

Apesar do termo *cluster* ser muito utilizado para descrever paralelismo, ele é mais bem descrito como um sistema distribuído. Diferente do paralelismo, que visa principalmente um método de execução de múltiplas instruções no mesmo intervalo de tempo dentro de uma única máquina, uma clusterização (múltiplos computadores) utiliza uma rede de conexão com diversas máquinas para o mesmo propósito. Há

três elementos básicos para a criação de um cluster: um aglomerado de computadores, uma rede de conexão entre eles e um programa que os permite compartilharem um trabalho pela rede (SLOAN, 2004). O *cluster* pode ser facilmente confundido com um sistema distribuído, pois possui características semelhantes, entretanto a diferença está principalmente no fato de que os nós dos *clusters* utilizam o mesmo sistema operacional e não podem ser endereçados individualmente (RAUBER; RÜNGER, 2010).

Os *clusters* podem ser divididos em dois grupos básicos, *clusters* simétricos e assimétricos. No *cluster* simétrico cada computador pode trabalhar individualmente, necessitando somente de uma rede para comunicação e os protocolos necessários, facilitando o acréscimo de um novo computador no *cluster*. Essa arquitetura é tipicamente classificada como *NOW* (*Network of Workstations*), em que cada computador pode trabalhar individualmente. Já o *cluster* assimétrico é comumente mais utilizado para *clusters* dedicados, em que os computadores (nós) do *cluster* necessitam de um computador mestre (cabeça) para comandar e comunicar entre o usuário e os nós, podendo minimizar os programas contidos nos nós, deixando somente o que é extremamente necessário. Como toda a informação que passa no *cluster* primeiramente precisa passar pelo mestre, o nível de segurança é maior, sendo necessária somente a manutenção do mestre (SLOAN, 2004).

O *cluster* pode ser dedicado, convencionalmente chamado de Beowulf (assim como na mitologia, é aquele que possui a força de muitos), e é desenvolvido de maneira a funcionar para aplicações mais específicas de forma mais otimizada (SLOAN, 2004). Os sistemas distribuídos não dedicados são utilizados diariamente sem que os usuários se dêem conta. Pode-se citar como exemplo a Internet, computação móvel, rede social Facebook, Youtube, entre outros, nos quais pessoas compartilham informações umas com as outras ao redor do globo terrestre (COULOURIS et al., 2007).

2.3 COMPLEXIDADE DE ALGORITMOS E META-HEURÍSTICAS

Em complexidade de algoritmos estudam-se as principais maneiras de classificar o quão difícil um algoritmo é, ou seja, medir questões como o tempo de resposta, quantidade de memória necessária e exatidão da resposta esperada pelo algoritmo. Em suas definições existem três principais que relacionam o tamanho de entrada de um problema com o tempo de solução do mesmo, essas são: melhor caso, pior caso e caso médio. A relação entre o tamanho de entrada de um problema e o tempo de execução gera uma função expressa pela denominada ordem assintótica que pode representar essencialmente três tipos de notações: Ω (big ômega), que define um limite inferior, *0* (big Oh), que define um limite superior e Θ (big theta), que define um limite exato, isto é, ao mesmo tempo inferior e superior (TOSCANI et al., 2012). As ordens assintóticas representam funções matemáticas bem conhecidas como: função polinomial, logarítmica, exponencial, fatorial, entre outras. A análise assintótica é uma maneira de definir os limites superiores e inferiores da função, sendo ela uma função que relaciona dados de entrada com o tempo para gerar a saída, ignorando os custos iniciais (MORET, 1997). Os problemas que possuem características de tempo polinomial (ordem assintótica superior $0(n^k)$, tal que $k \geq 0$) geralmente conseguem fornecer soluções em tempo de execução aceitável, sendo que o algoritmo acompanha de maneira polinomial as entradas de dados (CORMEN et al., 2002). Entretanto para problemas com características exponenciais ou mesmo fatoriais, não é possível obter uma solução em tempo aceitável. Para os problemas que são resolvidos em tempos polinomiais diz-se que são tratáveis, enquanto os que possuem tempos superpolinomiais como sendo intratáveis (CORMEN et al., 2002).

Os problemas que possuem características do tipo exponencial ou fatorial são tipicamente problemas de otimização que em sua maioria são classificados como NP-Completo (GAREY, 1979). Enquanto a classe NP-Completo visa determinar se o problema possui solução (sim ou não), a classe NP-Difícil visa determinar a solução para o problema. Apesar dos problemas NP-Completo e NP-Difícil não possuírem equivalentes polinomiais, é possível utilizar métodos para determinar respostas para o sistema, mesmo não sendo a melhor resposta possível. Esses métodos são conhecidos como meta-heurísticas, estratégias de alto nível

para que seja possível sair das respostas locais geradas pelos algoritmos mais simples (conhecidos como algoritmos gulosos) e atingir, se possível, uma resposta melhor (GLOVER; KOCHENBERGER, 2002). GRASP, Algoritmos Genéticos e Colônia de Formigas são bons exemplos de meta-heurística, sendo que cada uma possui sua única forma de tratar os dados (esses métodos e outros podem ser encontrados no livro *Handbook of Metaheuristics*) (GLOVER; KOCHENBERGER, 2002).

2.4 COMPLEXIDADE DE ALGORITMOS PARALELOS

O computador paralelo é aquele que consegue realizar múltiplas instruções em um mesmo período de tempo, tornando-o assim mais rápido do que o computador sequencial para certos problemas (SIPSER, 2007). A principal ideia da computação paralela é a troca do tempo de processamento por maior quantidade de computadores, ou seja, tempo por custo de máquina (MORET, 1997). E caso utilize-se “*n*” máquinas para resolver um problema, o seu tempo seria reduzido, no melhor dos casos, por “*n”*, sendo assim a paralelização não possui grande melhoria para resolução de problemas exponenciais, pois seu custo seria inviável. Moret (1997) destaca, ainda, que enquanto o estudo de complexidade sequencial se atém a classes de problemas que podem ser executados em tempos polinomiais, a complexidade paralela define uma classe de problemas que são resolvidos em um tempo sublinear com uma quantidade polinomial de máquinas.

Um dos modelos de circuito para computação paralela bem aceita é o circuito booleano, que é um grafo orientado acíclico utilizando os operadores lógicos “*E*”, “*OU*”, ou “*NÃO*” nos nós, com grau de entrada de tamanho constante, possuindo “*n*” entradas e “*m*” saídas, sendo que cada entrada possui ao menos um caminho para alguma saída, conforme Figura 1 – Modelo de circuito booleano (TERADA, 1990; MORET, 1997; SIPSER, 2007). Define-se também que o tamanho do circuito (quantidade de processadores) é o número de arestas e a profundidade é o maior caminho entre a entrada e a saída (o tempo de processamento). Para identificar quantos processadores são necessários para atingir uma determinada complexidade, ou vice-versa, utiliza-se a classe NC (classe de Nick, definida por

Stephen Cook) (SIPSER, 2007). A classe NC foi definida para os problemas que podem ser resolvidos em tempo $O(log^c n)$ e com $O(n^k)$ processadores (sendo que *c* e *k* são constantes), ou seja, são problemas decidíveis em tempo logarítmico com processadores polinomiais (TERADA, 1990; MORET, 1997; SIPSER, 2007) .

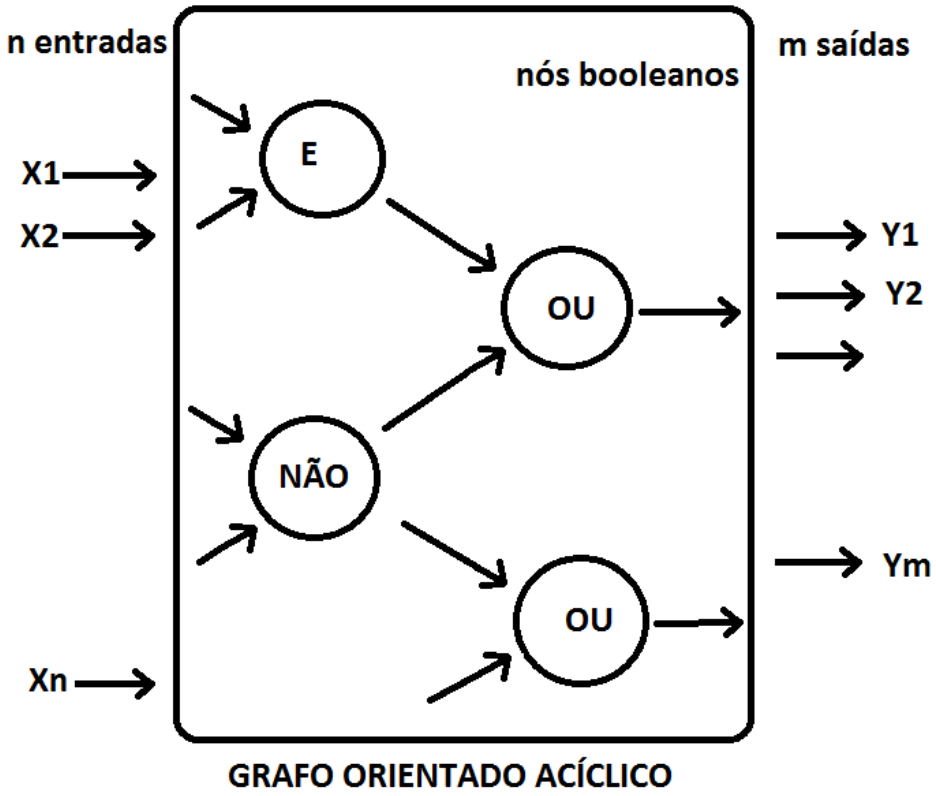

Figura 1 – Modelo de circuito booleano
Fonte: Terada (1990, p. 19).

2.5 O PROBLEMA DO CAIXEIRO VIAJANTE

Originalmente o Problema do Caixeiro Viajante – PCV (do inglês Traveling Salesman Problem – TSP) é um problema que visa determinar a menor rota possível para que um viajante possa percorrer '*n*' cidades, sem repeti-las, e, ao final, retornar a cidade de origem (JÜNGER; REINELT; RINALDI, 1994). A Figura 2 representa o Problema do Caixeiro Viajante, sendo os nodos 'A', 'B', 'C', 'D' e 'E' as cidades, e as arestas, os possíveis caminhos entre cidades com os valores das distâncias.

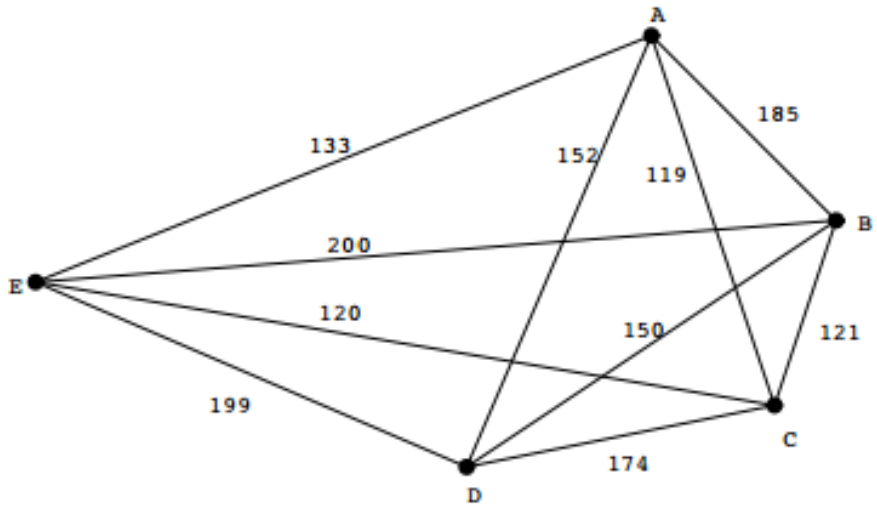

Figura 2 – Representação do Caixeiro Viajante
Fonte: http://arquivoescolar.org/bitstream/arquivo-e/45/3/metodos_finitos_III.pdf
Acessso em: 1de agosto de 2015.

O PCV é considerado simétrico quando é possível se mover para qualquer cidade independentemente da direção escolhida (JÜNGER; REINELT; RINALDI, 1994).

2.6 SPEEDUP

O cálculo de *speedup* é uma métrica para comparar algoritmos paralelos com algoritmos sequenciais que foi definido primeiramente pela lei de Amdahl. A lei de Amdahl define que sempre haverá uma parte do programa que não será paralelizável, dessa forma não é possível obter ganho infinito ao paralelizar algum problema (NOBRE, 2011). O *speedup* visa comparar o algoritmo paralelo desenvolvido com o melhor algoritmo sequencial disponível até o momento, podendo assim dizer se o algoritmo paralelo realmente é melhor. O *speedup* pode ser definido pela Equação (1).

$$Speedup = \frac{T_{sequencial}}{T_{paralelo}} \quad (1)$$

Dessa forma obtém-se uma quantificação direta do quão melhor é um algoritmo paralelo em comparação ao sequencial, ou seja, $T_{paralelo}$é o valor do $Speedup$ vezes mais rápido que o $T_{sequencial}$. Apesar de ser uma métrica para comparação, a lei de Amdahl não leva em consideração o tempo de sincronização

para os algoritmos paralelos, podendo resultar em predições pouco realísticas para alguns problemas (NOBRE, 2011).

2.7 HP LINPACK

HP Linpack, ou mesmo HPL (*High Performance Linpack*), é um software livre, desenvolvido pelo departamento de Ciência da Computação na Universidade do Tennessee, que visa realizar cálculos de sistemas lineares randômicos com precisão dupla (64 bits). O pacote HPL provê um programa para testar e temporizar a precisão das soluções obtidas (quantificadas em GFLOPS), sendo necessária a biblioteca de comunicação MPI e uma implementação das seguintes bibliotecas: *Basic Linear Algebra Subprograms* (BLAS) ou *Vector Signal Image Processing Library* (VSIPL). Apesar de algumas restrições hipotéticas da rede de interconexão utilizada, o algoritmo utilizado neste pacote é escalável, ou seja, sua eficácia paralela é mantida constante em relação ao uso de memória por processador (PETITET; WHALEY; DONGARRA; CLEARY, 2014).

2.8 MICROCONTROLADORES

Microcontroladores, também chamados de microcomputador de um só chip (*SoC – System-on-a-chip*), são sistemas sequenciais que utilizam passos para realizarem funções específicas e reúnem diversos elementos como RAM, ROM, temporizadores, canais de comunicação, conversores, portas de I/O, etc. em um único chip (SILVA JUNIOR, 1998).

2.8.1 Conversor Analógico/Digital

Analog to Digital Converter (*ADC*), ou Conversor Analógico/Digital, é um elemento que transforma sinais analógicos em digitais. Uma quantidade analógica pode assumir qualquer valor dentro de um intervalo, de forma que seu valor exato é significativo. Em contrapartida uma quantidade digital pode ser descrita entre duas

possibilidades, tal como 0 ou 1, verdadeiro ou falso, baixo ou alto, etc., que fisicamente pode ser uma faixa de valores analógicos. Por exemplo, a saída de um sensor analógico pode ser uma tensão de 2,76V, que convertida para digital apresenta um valor de 27,6ºC. A maioria das variáveis físicas é analógica e podem assumir qualquer valor entre um intervalo contínuo de valores, assim como: luz, som, temperatura, velocidade, tensão, corrente, entre outras, e para introduzi-las ao sistema digital é necessário realizar uma conversão para digital (TOCCI; WIDMER, 2000).

Para que seja possível converter um sinal analógico para digital, primeiramente é necessário converter o tipo de sinal analógico para sinais elétricos, caso o sinal já não seja elétrico, e para isso são utilizados transdutores. Os transdutores geram saídas elétricas (corrente ou tensão) proporcionais a variável física que está monitorando, por exemplo, a temperatura de um recipiente de água que varia entre 10 a 20ºC pode ser condicionada a uma saída de 10 a 20mV. Após o sinal ser condicionado é possível utilizar um conversor de analógico para digital, podendo converter o exemplo anterior para uma faixa entre 1010_2 (10_{10}) a 10100_2 (20_{10}) (TOCCI; WIDMER, 2000).

O passo de quantificação é o responsável por determinar o valor de cada uma das amostras recebidas, sendo que a resolução ou sensibilidade da quantificação está relacionada diretamente com o fundo de escala e o número de *bits* de cada conversor. De acordo com Fialho (2004), a resolução ou sensibilidade de um conversor é a mínima variação de sinal analógico que provocará uma mudança, superior ou inferior, do valor da saída, podendo ser definida como mostra a Equação 2.

$$S = \frac{1}{2^N} \cdot FE \tag{2}$$

Sendo que:

- S: sensibilidade do conversor A/D;
- N: número de *bits* disponíveis para quantificar;
- FE: fundo de escala do conversor.

2.8.2 Transmissor/Receptor Assíncrono Universal

Também conhecido como UART (*Universal Asynchronous Receiver/Transmiter*), é um modo de comunicação que permite o microcontrolador trocar informações com computadores, microcontroladores, dispositivos de comunicação, etc. (PEREIRA, 2005). Neste modo de comunicação não há necessidade de haver um sincronismo entre emissor e receptor, sendo que cada caractere é transmitido individualmente. Neste modo a comunicação é iniciada por *bits* de início de transmissão (*Start bit*) e finalizada por *bits* de término de transmissão (*Stop bits*) (SILVA JUNIOR, 1998).

Quando o *Start bit* é reconhecido pelo sistema, o *clock* interno realiza uma varredura no canal de dados de tempos em tempos para detectar seu nível e determinar seu *bit*, zero ou um. Ao reconhecer o último *bit* de dados, o sistema fica a espera dos *Stop bits* para finalizar a transmissão dos dados e voltar ao estado de espera. Para este tipo de comunicação é necessário que ambos, transmissores e receptores, operem numa mesma taxa de transmissão/recepção, geralmente quantificada em BAUD (SILVA JUNIOR, 1998; PEREIRA, 2005).

2.9 EFEITO HALL

O efeito Hall tem este nome devido ao nome da pessoa que o descobriu em 1879, o físico Edwin. H. Hall, cujo experimento comprovou que o surgimento de uma força eletromotriz (*f.e.m.*) proporcional a, o produto do campo magnético $\vec{B}$ perpendicular e à velocidade dos elétrons v. Apesar de ser descoberto em 1879, suas aplicações técnicas só se tornaram possíveis em meados de 1950, com a descoberta de alguns semicondutores capazes de suportar altas velocidades de

elétrons em menores quantidades, necessário para mensuração da tensão Hall, que chega a uma ordem de *100 mV* (BALBINOT; BRUSAMARELLO, 2007).

Campos magnéticos e induções magnéticas estão associados a corrente elétrica, ou seja, a medição de um campo magnético está relacionado indiretamente a uma corrente elétrica. Uma vez que o campo magnético $\vec{B}$ é exatamente proporcional à corrente i, a medida de corrente pode ser efetuada sem a necessidade de abrir o circuito ou realizar contatos desnecessários entre o condutor e o instrumento de medição (BALBINOT; BRUSAMARELLO, 2007).

2.10 VALOR EFICAZ

Valor eficaz, ou RMS (*Root Mean Square*), é uma função que representa o trabalho efetivo de uma grandeza variável no tempo entre excursões positivas e negativas de uma função. O valor eficaz de uma função discreta é dado pela raiz quadrada do somatório dos quadrados dos valores dividido pelo número de eventos (Equação 3) (MUSSOI, 2006).

$$V_{ef} = \sqrt{\frac{\sum_{i=0}^{n} (V_i)^2}{n}} \tag{3}$$

Sendo que:

- V_{ef}: valor eficaz;
- n: número de eventos;
- V_i: amostra da grandeza.

Já o valor eficaz de uma função contínua é dado pelo cálculo da média quadrática através do uso da integral descrita na Equação 4 (MUSSOI, 2006).

$$V_{ef} = \sqrt{\frac{1}{T}\int_{t_i}^{t_f} v(t)^2 . dt} \quad (4)$$

Sendo que:

- V_{ef}: valor eficaz;
- T: período de tempo;
- t_i: tempo inicial;
- t_f: tempo final;
- $v(t)$: função da grandeza.

De acordo com Mussoi (2006, p. 32) “o valor da tensão eficaz ou corrente eficaz de uma forma de onda é o valor matemático que corresponde a uma tensão ou corrente contínua constante que produz o mesmo efeito de dissipação de potência numa dada resistência”.

3 MATERIAIS E MÉTODOS

Neste capítulo são abordados os materiais necessários para construção de ambos os *clusters* (*desktop* e ARM), assim como os métodos utilizados para teste e coleta de dados.

3.1 MATERIAIS

Para este projeto foram utilizados computadores *desktop* tradicionais e alguns dispositivos baseados na tecnologia ARM. As especificações dos computadores *desktop* e dispositivos ARM são apresentados nos Quadros 1, 2 e 3, respectivamente.

Modelo	HP Compaq 6005 Pro Microtower
Processador	*DualCore* AMD Phenom™ II X2 B53
Clock	2,8 GHz
Memória RAM	8 GB
Disco Rígido	300 GB
Fonte de alimentação	HP Power Supply (PS-4321-9HP)
Vmain	12,1V / 16 A (193,6 W)
Vcpu	12,1 V / 14 A (169,4 W)
Potência máxima da fonte	320W

Quadro 1 – Especificações computadores *Desktop* HP
Fonte: Autoria própria.

Modelo	Raspberry Pi Modelo B
Processador	ARM1176JZF-S *core*
Clock	700 MHz
Memória SDRAM	512 MB
Memória Flash	4 GB
Fonte de alimentação	I.T.E. Power Supply
Potência máxima consumida	5W (5V, 1A)

Quadro 2 – Especificações Raspberry Pi, Modelo B
Fonte: Autoria própria.

Modelo	Cubietruck
Processador	ARM Cortex-A7
Clock	1 GHz dual-core
Memória DDR3	2 GB
Memória Flash NAND	8 GB
Fonte de alimentação	Portas USB
Potência consumida	Entre 5W (5V, 1A) e 12,5W (5V, 2,5A)

Quadro 3 – Especificações CubieTruck
Fonte: Autoria própria.

O Quadro 4 apresenta os sistemas operacionais, *switches* e quantidade de equipamentos utilizados tanto para o *cluster desktop*, quanto para o *cluster* ARM.

	Cluster Desktop	*Cluster* Raspberry	*Cluster* Cubietruck
Sistema Operacional	Ubuntu Server	Raspbian Wheezy	Linaro Lubuntu Server
Versão	14.04 LTS	2014-06-20	13.06
Switch	Cisco SG500 24 Portas	Cisco Catalyst 2960 24 Portas	Cisco Catalyst 2960 24 Portas
Quantidade de equipamentos	7 Computadores	8 Dispositivos	8 Dispositivos

Quadro 4 – Sistemas e componentes utilizados
Fonte: Autoria própria.

Além desses componentes, foi utilizada a biblioteca de troca de mensagens MPI, o *benchmark* HP Linpack e cabos de comunicação RJ-45 CAT.6.

Os preços de cada equipamento situam-se no Quadro 5, pesquisados no dia 25 de abril de 2015.

Equipamento	Valor	Site
Computador Desktop	R$ 1.498,00	http://walmart.com.br/marca/positivo/153
Raspberry Pi B	R$ 289,79	http://lojamundi.com.br/
Cubietruck	R$ 506,99	http://lojamundi.com.br/
Fonte 5W/10W	R$ 24,02	http://lojamundi.com.br/
Cisco SG500	R$ 4.100,39	https://bestmarket.com.br/
Cisco Catalyst 2960	R$ 2.199,01	https://bestmarket.com.br/
RJ-45 CAT.6 5m	R$ 17,50	http://cirilocabos.com.br/

Quadro 5 – Coração dos equipamentos
Fonte: Autoria própria.

Como o computador utilizado neste trabalho não se encontrava disponível no mercado, contabilizou-se pelo equipamento mais parecido disponível. O computador contabilizado foi o Positivo Dri7432 com processador Intel Core i3 e 8GB de memória RAM, disponível no site: http://www.walmart.com.br/marca/positivo/153.

Para calcular o preço de cada *cluster* foi desconsiderado o preço do cabeamento, pois isso é dependente da distância entre o *switch* e os dispositivos. O custo total de cada *cluster*, Tabela 1, é contabilizado pela quantidade de dispositivos, *switch* e fontes utilizados.

Tabela 1 – Custo em R$ de cada *cluster*

Cluster	***Desktop***	**Raspberry**	**Cubietruck**
Total	R$ 14.586,39	R$ 4.709,49	R$ 6.447,09

Fonte: Autoria própria.

Para o sistema de medição de energia utilizado neste trabalho, foram utilizados os seguintes componentes disponíveis:

- Computador, para recepção de dados;
- Microcontrolador Tiva TM4C123GH6PM;
- Sensor de corrente por Efeito Hall ACS714;
- 2 resistores (*7,5k* e *3,9k*), para o divisor de tensão;
- 1 capacitor de *2u*, necessário para o divisor de tensão;
- No Break Power Vision II SMS.

Como não é utilizado nenhum sensor de medição de tensão para saber as possíveis variações provenientes das redes de alimentação, é possível diminuir o erro de medição utilizando um *No Break*, já que ele tende garantir uma saída de tensão uniforme.

Para o desenvolvimento deste trabalho foram utilizadas as ferramentas apresentadas no Quadro 6, sendo que a única ferramenta não gratuita utilizada, Microsoft Visio, foi disponibilizada aos alunos, pela UTFPR.

Ferramenta	Objetivo
CodeBlocks	Desenvolver os algoritmos sequenciais e paralelos
Code Composer Studio	Desenvolver o algoritmo e programar o microcontrolador Tiva
Netbeans	Desenvolver o programa Java para comunicação e coleta de dados
ObjectAid UML Explorer para Eclipse	Criar diagrama de classes disposto na Figura 12
Microsoft Visio 2013	Criar as imagens e fluxogramas utilizados neste trabalho
Notepad++	Exportar os algoritmos desenvolvidos em formato adequado para leitura

Quadro 6 – Ferramentas utilizadas
Fonte: Autoria própria.

3.2 MÉTODOS

Três *clusters* homogêneos foram formados com: computadores *desktop*, Raspberry Pi e Cubietruck. Cada *cluster* é composto por um dispositivo “mestre” que é encarregado de controlar os nodos “escravos”, conforme a Figura 3.

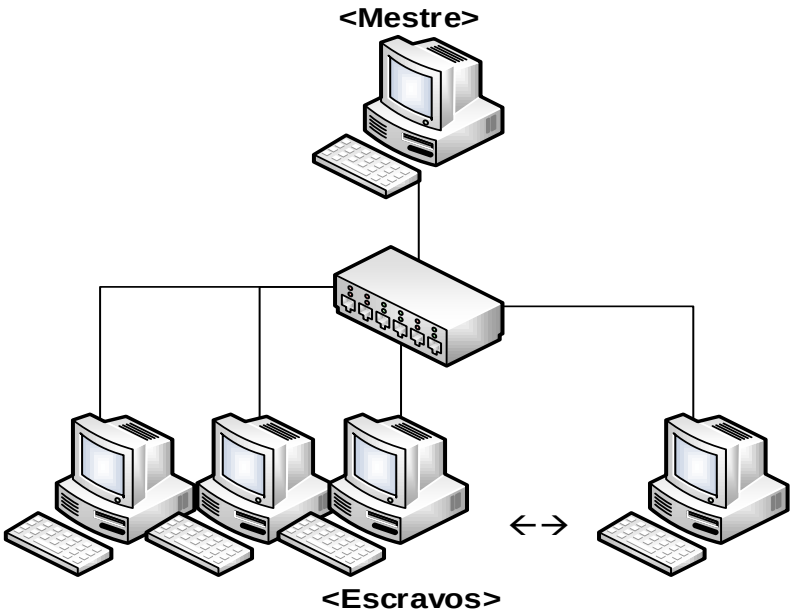

Figura 3 – Representação dos *clusters*
Fonte: Autoria própria.

Após os ambientes estarem montados, foi instalada uma versão Linux em cada máquina com os requisitos mínimos para seu funcionamento, assim como a bibliotecas MPI e HP Linpack. Como foi utilizada a biblioteca MPI, os algoritmos implementados utilizaram a linguagem de programação C, desenvolvidos tanto para computação sequencial, quanto para computação paralela. Os problemas utilizados são:

- Multiplicação de Matriz *mxn;*
- Caixeiro Viajante.

As informações de tempo de execução dos algoritmos testados nesses ambientes são armazenadas para realização dos cálculos de *speedup*. As respostas retornadas dos escravos são processadas no mestre, para que as mesmas possam ser testadas e validadas.

Os tempos de execução e custo energético foram coletados durante a execução do algoritmo e os custos calculados por meio dos dados fornecidos pelos fabricantes e sites de compra. Com as respostas dos algoritmos e os dados coletados a partir dos fabricantes, calcula-se o custo/benefício de cada cluster e mostrados no Capítulo 4 deste documento.

As configurações do HP Linpack, que são dadas pelo arquivo “HPL.dat”, contém as informações necessárias para executar o teste, como: quantidade de

problemas, tamanho dos problemas, números de blocos, tamanho dos blocos, etc.. Para obter o melhor desempenho possível, estes dados devem ser definidos diferentemente para cada tipo de *cluster*, visando o tipo de arquitetura utilizada. De forma a se obter dados corretos para cada tipo de *cluster*, foram utilizadas ferramentas online de otimização para HP Linpack, como: “HPL Calculator” (SINDI, 2015) e “How do I tune my HPL.dat file?” (ADVANCED CLUSTERING TECHNOLOGIES, 2015).

Para estimar a energia gasta nos *clusters* (não contabilizando *switches*), foi utilizado o microcontrolador da família Tiva para conversão e temporização dos dados coletados a partir do sensor de corrente ACS714, representado pelo esquema lógico na Figura 4.

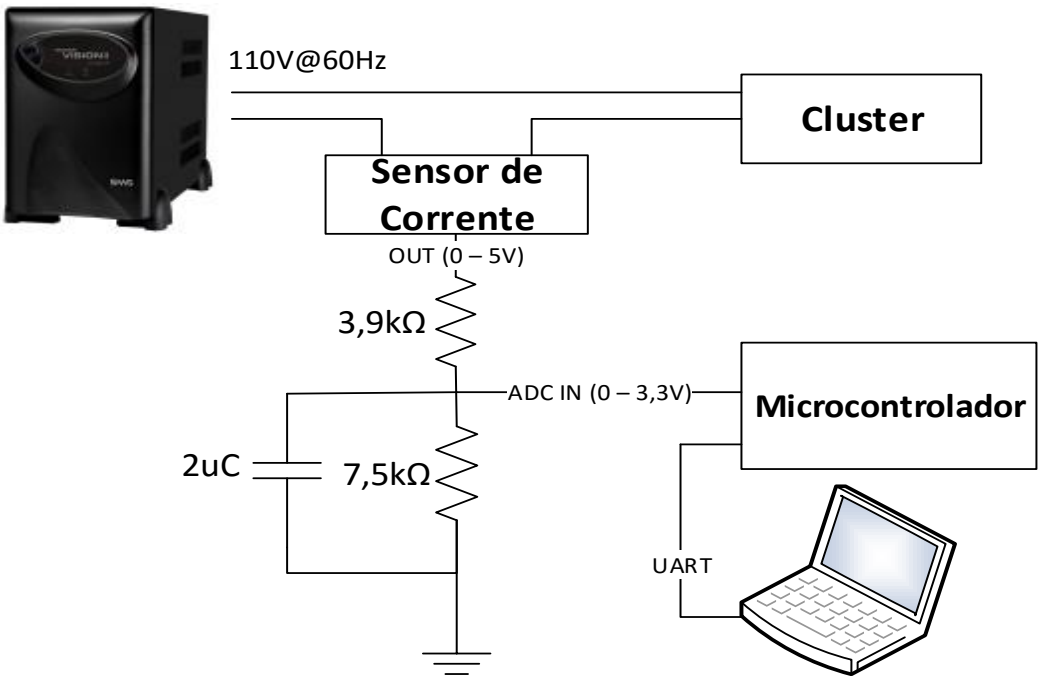

Figura 4 – Esquema lógico para captura de energia.
Fonte: Autoria própria.

Para converter a corrente alternada em corrente contínua foram utilizadas 200 amostras para cálculo da corrente RMS, sendo que para cada amostra foi utilizada uma média de 64 amostras, ou seja, 12.800 amostras por ciclo. Como a saída do *No Break*, assim como o sistema de energia convencional, utilizam uma frequência de *60Hz* para alimentação, ou seja, o ciclo para conversão do ADC, condicionando assim uma taxa de amostragem de 768.000 amostras por segundo,

logo abaixo do limite de um milhão de amostras por segundo que esse conversor proporciona.

Como a saída do sensor varia entre *0* e *5V*, e a entrada do ADC do microcontrolador Tiva suporta somente entre *0* a *3,3V* (fundo de escala), foi utilizado um divisor de tensão com fator aproximado de *0,65789*, transformando a saída do sensor de corrente utilizado de *66mV/A* (ALLEGRO MICROSYSTEMS, 2014) para uma escala de aproximadamente *43,421mV/A*. A partir do fundo de escala estipulado e, sabendo que o conversor utilizado é de 12 bits, ao utilizar a Equação 2 obtêm-se uma sensibilidade de *0,8059mV* para o ADC. Sabendo-se a escala do sensor e a sensibilidade de conversor, define-se que a sensibilidade do sensor de corrente é de *18,559mA ou 2,04149W* para um tensão de *110V*.

4 DESENVOLVIMENTO

Neste capítulo estão dispostos os elementos desenvolvidos, necessários para este trabalho, como: descrições do sistema, métodos de comunicação, implementações de algoritmos e configurações dos *clusters*.

4.1 DESCRIÇÃO DO SISTEMA

O sistema utilizado para comparar o *cluster desktop* com o *cluster* ARM utiliza códigos em linguagem de programação C e a biblioteca HP Linpack para estimar o desempenho, além de um sistema para realizar a medição de custo energético durante o processamento dos códigos. O sistema para medição de energia é composto por um sensor de corrente operante por Efeito Hall (ACS714), tendo sua saída em tensão proporcional a corrente mensurada. Esta tensão analógica é devidamente convertida e enviada a uma central de dados, nomeada Central, que realiza os devidos procedimentos e interpretações.

A Central opera com um programa codificado em Java, nomeado JCentral, que utiliza bibliotecas de comunicação para que sejam realizadas as devidas comunicações com o microcontrolador e o *cluster*. Foi utilizada a biblioteca RXTXSerial para realizar comunicação com o microcontrolador através da UART, enquanto para a comunicação com o mestre do *cluster* foi utilizada a biblioteca JSch (*Java Secure Channel*) que opera com o protocolo SSH (*Secure Shell*). Com isso é possível realizar o sincronismo de início e fim do processamento do *cluster* com o início e fim da coleta de energia pelo microcontrolador. Após terminar a coleta de dados, a Central armazena as medidas recebidas em um documento contendo o código executado, o tempo total em milissegundos (*ms*) e a energia gasta em watts segundo (*Ws*). A Figura 5 apresenta as interconexões de comunicação deste sistema.

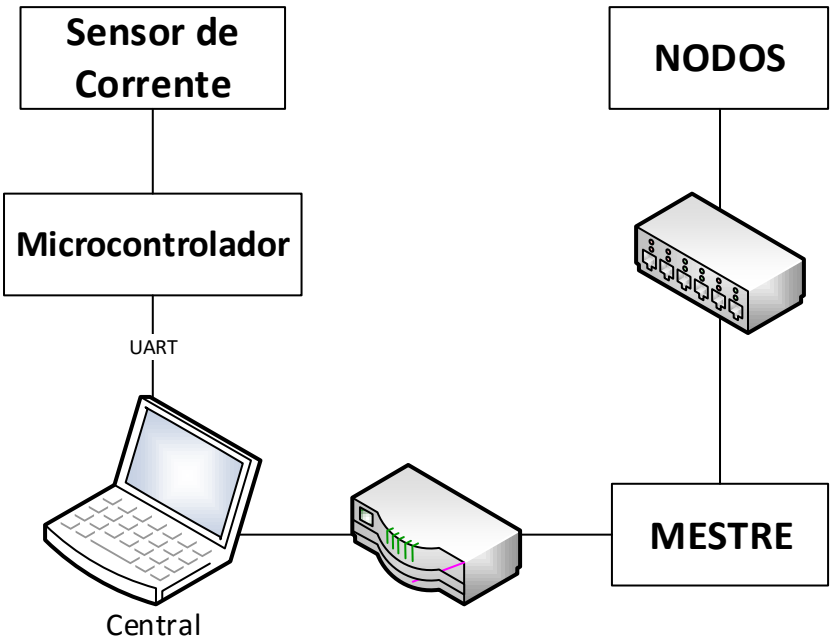

Figura 5 – Descrição da rede de comunicação entre dispositivos.
Fonte: Autoria própria.

Os sistemas utilizados para testar o desempenho dos *clusters* foram implementados para execução sequencial e paralela, sendo utilizada a biblioteca MPI para paralelização e distribuição do algoritmo aos *clusters*. A topologia utilizada nos *clusters* foi de mestre/escravo, ou seja, a função do mestre é de início de programa, sincronismos entre os nodos e coleta das respostas, enquanto os nodos realizam as funções de processamento. Como as arquiteturas, utilizadas neste trabalho, variam de processadores com ou sem múltiplos núcleos, foi definido que os dispositivos mestres iriam executar com um único processo, enquanto os nodos escravos iriam executar com a quantidade disponível de núcleos para cálculos, necessitando ao menos de dois dispositivos.

4.2 APRESENTAÇÃO

Para acessar os *clusters*, configurá-los, transferir arquivos, etc., foram utilizado os programas PuTTY (Figura 6 e Figura 7) e WinSCP (Figura 8) que utilizam o protocolo SSH de comunicação.

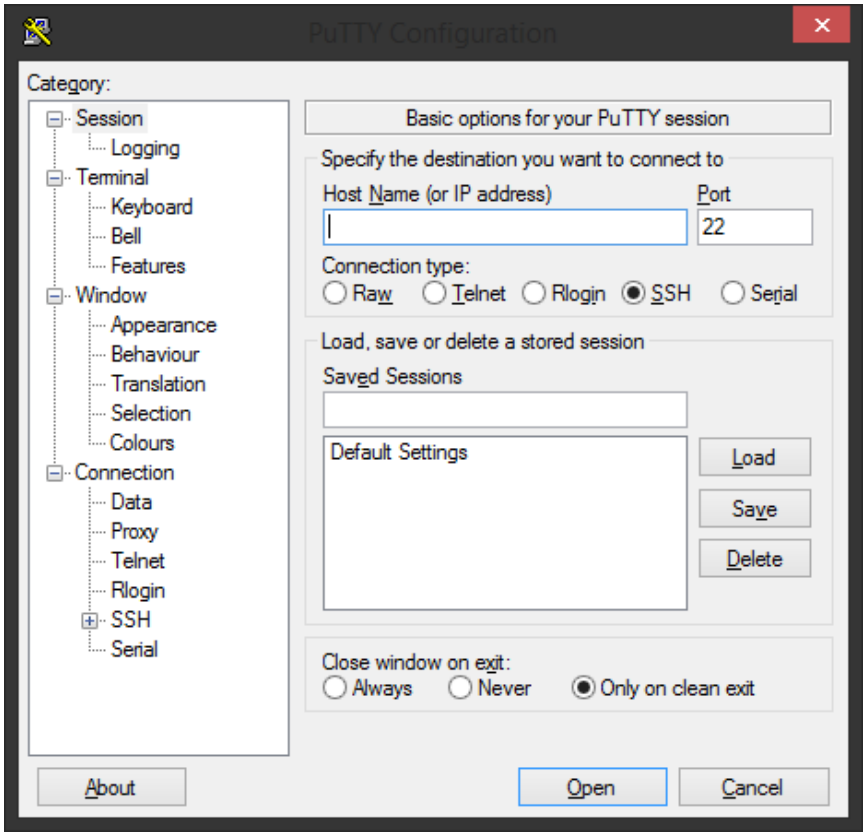

Figura 6 – Tela PuTTY
Fonte: Autoria própria.

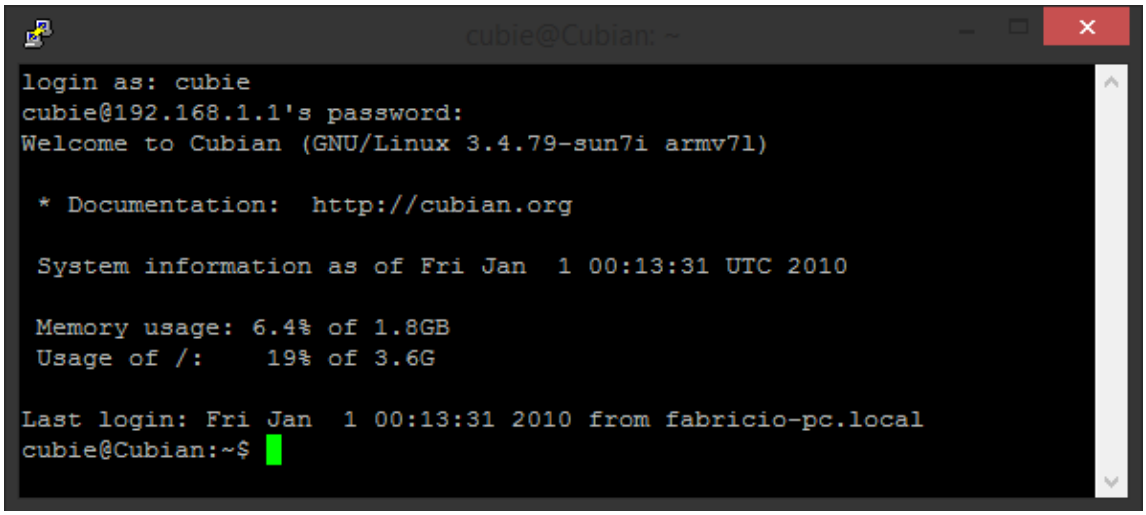

Figura 7 – Terminal de comunicação do PuTTY
Fonte: Autoria própria.

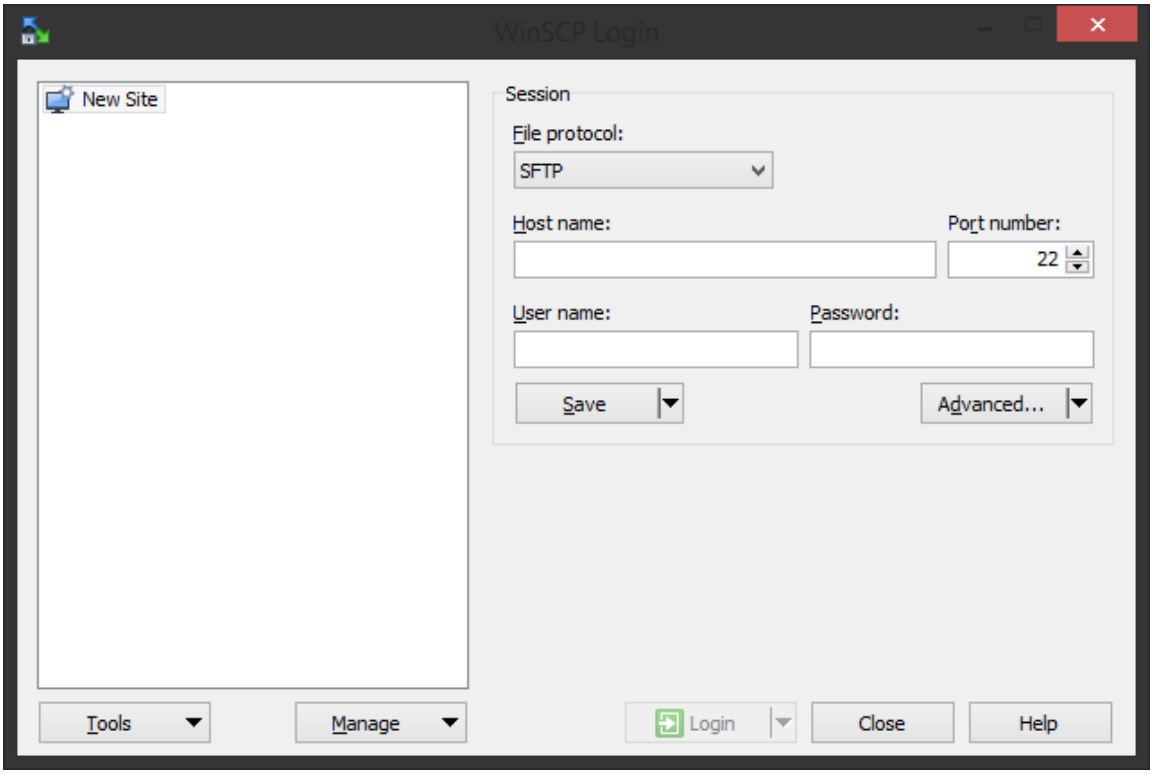

Figura 8 – Tela WinSCP
Fonte: Autoria própria.

Visto que os sistemas operacionais utilizados nos *clusters* não possuem interface gráfica e são acessados remotamente, é necessário realizar chamadas de comando por terminal para executar os programas de testes. O Quadro 7 demonstra como utilizar os comandos por terminal.

Multiplicação de matriz sequencial: time ./MM_SEQ ./MATRIZ_A ./MATRIZ_B
Caixeiro Viajante sequencial: time ./CV_SEQ ./ARQUIVO_ENTRADA
Multiplicação de maitrz com MPI: time mpiexec -f machinefile -n QTD_PROC ./MM_MPI ./MATRIZ_A ./MATRIZ_B
Caixeiro Viajante com MPI: time mpiexec -f machinefile -n QTD_PROC ./CV_MPI ./ARQUIVO_ENTRADA
HP Linpack: mpiexec –f machinefile –n QTD_PROC ./xhpl ./HPL.dat

Quadro 7 – Como executar os programas

Nos quais:

- *time* é a função do Linux que retorna o tempo gasto para processar;
- MM_SEQ e MM_MPI são os programas de multiplicação de matriz sequencial e paralelo, respectivamente;
- MATRIZ_A e MATRIZ_B são os arquivos que contêm as matrizes para multiplicação AxB;
- xhpl é o programa executável da biblioteca HP Linpack;
- CV_SEQ e CV_MPI são os programas de resolução do Caixeiro Viajante sequencial e paralelo, respectivamente;
- ARQUIVO_ENTRADA é o arquivo que contém informações necessárias para gerar um grafo completo para o problema do Caixeiro Viajante;
- HPL.dat é o arquivo de configuração para o executável do HP Linpack;
- mpiexec é a diretiva de execução da biblioteca MPI;
- machinefile é o arquivo contendo as informações dos dispositivos que participarão dos cálculos;
- QTD_PROC é a quantidade de processos que serão gerados.

Já o programa utilizado para fazer a sincronia entre execução e captura de dados, JCentral, foi desenvolvido em Java utilizando a interface gráfica apresentadas nas Figuras 9 e 10.

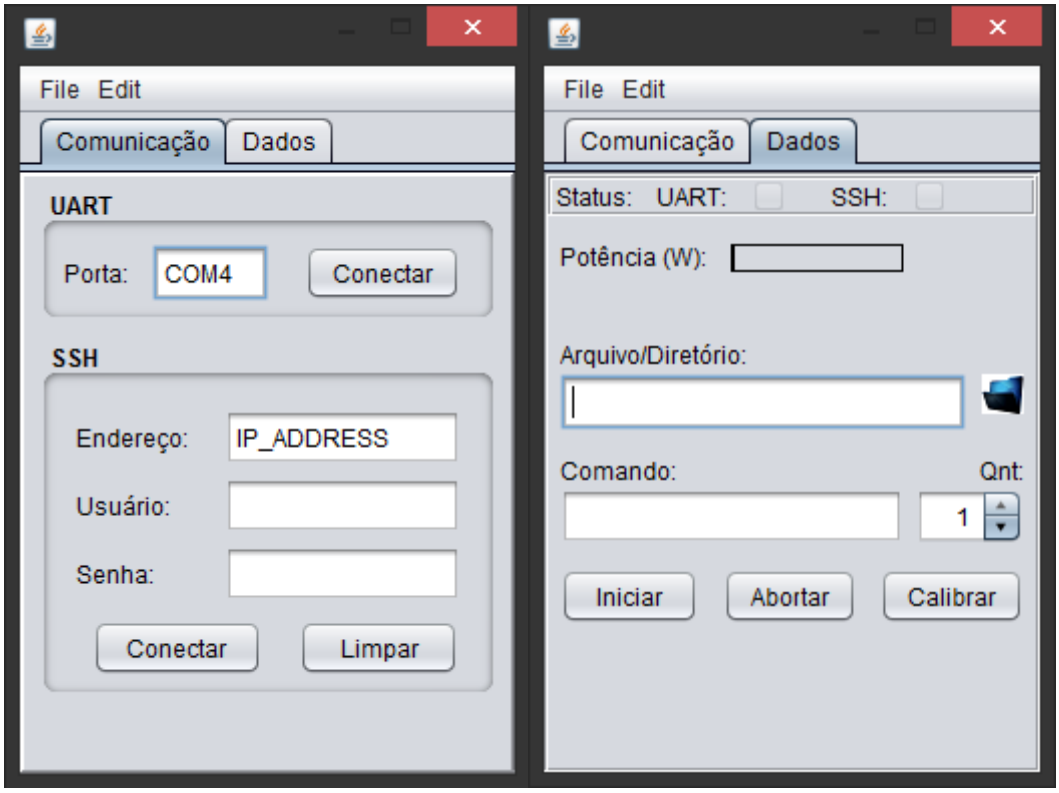

Figura 9 – Telas de comunicação, execução e captura de dados.
Fonte: Autoria própria.

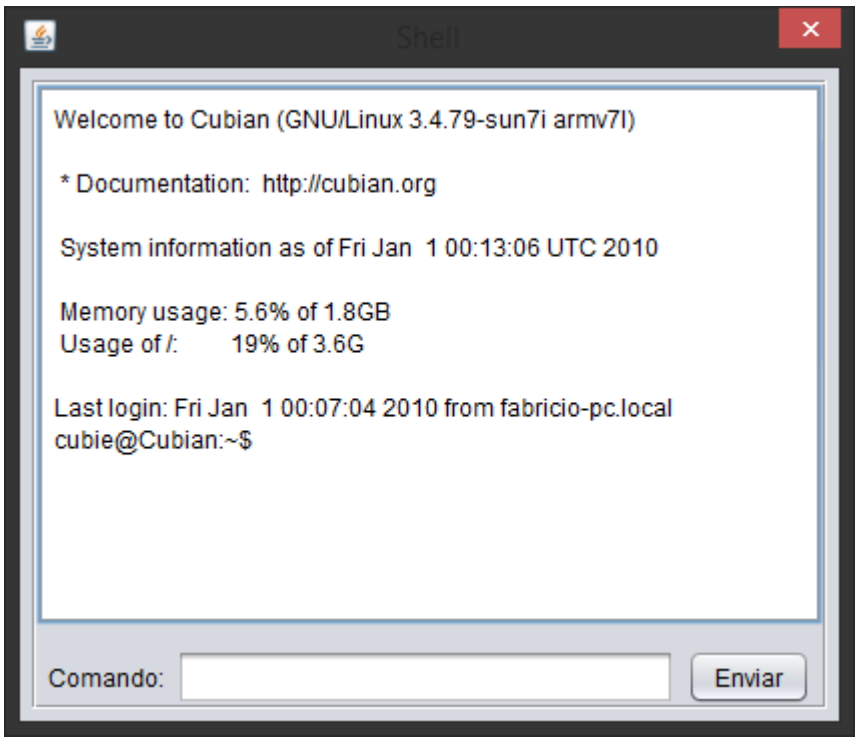

Figura 10 – Terminal Java.
Fonte: Autoria própria.

Na aba “Comunicação”, mostrado na Figura 9, estão todos os campos necessários para realizar a comunicação entre a Central e os dispositivos. Para conectar ao microcontrolador, é necessário selecionar a porta COM disponível no ambiente Windows e realizar a conexão. Para a conexão SSH com o *cluster* é necessário informar o IP do mestre, assim como a conta para conexão. Ao realizar a conexão com o *cluster*, será aberta outra janela (Figura 10) que contém um terminal de comunicação com *cluster*, na qual comandos podem ser digitados na linha inferior, e executados com o botão “Enviar”.

Depois de estabelecida a comunicação com o microcontrolador e o *cluster*, na aba “Dados”, mostrado na Figura 9, serão atualizados os campos “UART” e “SSH”, mostrando que as conexões ocorreram bem sucedidas. Para iniciar a captura de dados e a execução do programa, é necessário informar o local e nome do arquivo no qual serão salvos os dados no campo “Arquivo/Diretório”. Ao configurar o comando e a quantidade de vezes que será executado, basta pressionar o botão “Iniciar” para começar a análise do *cluster*. Enquanto o programa fica em execução, o campo “Potência” é atualizado a cada segundo com a potência mensurada. Os botões “Abortar” e “Calibrar” servem respectivamente para parar o que está sendo executado, e gerar um arquivo com as informações vindas do sensor, sem a necessidade de uma comunicação SSH ou comando.

4.3 IMPLEMENTAÇÃO

Os algoritmos implementados neste trabalho e as demais configurações necessárias para os ambientes dos *clusters* estão descritos nas subseções a seguir.

4.3.1 Multiplicação de Matriz

O programa de multiplicação de matriz, sequencial e paralelo, utilizam arquivos contendo matrizes, sendo que a primeira e segunda linha do arquivo devem ser um número referente à quantidade de linhas (“*m*”) e colunas (“*n*”), respectivamente, da matriz. A seguir, deverá ter “*m*” linhas, cada uma contendo “*n*” elementos separados por ponto e vírgula, sendo que o elemento pode ser

fracionário, utilizando o ponto para separar a casa decimal. O Anexo A apresenta um exemplo de entrada para o programa de multiplicação de matriz.

Para solucionar a multiplicação de matriz, foi utilizada a metodologia de multiplicação entre linhas e colunas, somando-as ao final, sendo a ordem assintótica superior do algoritmo sequencial $O(n^3)$. O Quadro 8 apresenta um pseudocódigo de multiplicação de matriz sequencial.

```
PROGRAMA Multiplicacao_de_matriz_sequencial

VARIAVEIS
i, j, k: INTEIRO;
LinhaA, ColunaA, LinhaB, ColunaB: INTEIRO;
MatrizA, MatrizB, MatrizC: REAL;

INICIO
LEIA(MatrizA);
LEIA(MatrizB);

PARA i <- 0 ATE LinhaA PASSO 1 FACA
    PARA j <- 0 ATE ColunaB PASSO 1 FACA
        MatrizC[i][j] <- 0;
        PARA k <- 0 ATE ColunaA PASSO 1 FACA;
            MatrizC[i][j] <- MatrizC[i][j]+(MatrizA[i][k]*MatrizB[k][j]);
        FIMPARA
    FIMPARA
FIMPARA

FIM.
```

Quadro 8 – Multiplicação de matriz sequencial
Fonte: Autoria própria.

Para realizar a paralelização deste método o nodo mestre do *cluster* realiza a leitura da primeira matriz para divisão entre os demais nodos. Os nodos escravos realizam a leitura da segunda matriz e, juntamente com os dados provenientes do mestre, realizam os métodos de multiplicação. Ao finalizarem a multiplicação, todos os dados dos nodos escravos são retornados ao nodo mestre, finalizando o programa. O Quadro 9 demonstra como o processo é realizado.

```
PROGRAMA Multiplicacao_de_matriz_paralela

VARIAVEIS
MESTRE <- 0: INTEIRO;
MPI_ID, QTD_MPI_ID: INTEIRO;
i, j, k: INTEIRO;
LinhaA, ColunaA, LinhaB, ColunaB: INTEIRO;
MatrizA, MatrizB, MatrizC: REAL;

INICIO
```

```
INICIAR_MPI();
    // Se for o mestre, realizar função de sincronização
    SE MPI_ID = MESTRE ENTAO
        LEIA(MatrizA);
        // Enviar linha da matriz A aos nodos
        PARA i <- 0 ATE LinhaA PASSO 1 FACA
            ENVIE_MPI(MatrizA[i], i%QTD_MPI_ID+1);
        FIMPARA

        // Receber resultados
        PARA i <- 0 ATE LinhaA PASSO 1 FACA
            RECEBE_MPI(MatrizC[i], i%QTD_MPI_ID+1);
        FIMPARA

    // Senão, realizar função de cálculos
    SENAO
        LEIA(MatrizB);
        PARA i <- MPI_ID ATE LinhaA PASSO i <- i+QTD_MPI_ID FACA
            // Receber linha da matriz A do mestre
            RECEBE_MPI(MatrizA[i], MESTRE);
            // Realizar multiplicação
            PARA j <- 0 ATE ColunaB PASSO 1 FACA
                MatrizC[i][j] <- 0;
                PARA k <- 0 ATE ColunaA PASSO 1 FACA
         MatrizC[i][j] <- MatrizC[i][j] + (MatrizA[i][k] * MatrizB[k][j]);
                FIMPARA
            FIMPARA
            // Enviar linha da matriz C ao mestre
            ENVIE_MPI(MatrizC[i], MESTRE);
        FIMPARA
    FIMSE
FIM.
```

Quadro 9 – Multiplicação de matriz paralela
Fonte: Autoria própria.

4.3.2 Caixeiro Viajante

O algoritmo utilizado para solucionar o problema do Caixeiro Viajante Simétrico é composto por um arquivo de entrada de dados, cujos exemplos foram retirados do site TSPLIB da Universidade de Heidelberg (REINELT, 2015). O arquivo de entrada é composto pelo número de cidades “*n*”, em sua primeira linha, seguido por “*n*” linhas contendo o identificador da cidade e sua posição cartesiana “*x,y*”, ordenado pelo identificador da cidade. O Anexo B apresenta um exemplo de entrada para o problema do Caixeiro Viajante.

O algoritmo sequencial, apresentado no Quadro 10, é composto por duas partes essenciais, a construção de uma solução inicial e o aprimoramento por métodos aleatórios. Para construir a solução inicial foi utilizado o Método Guloso (FEOFILOFF, 2015) por distância entre cidades, de forma que a melhor solução

gulosa selecionada foi proveniente da alteração da cidade na qual o caixeiro inicia sua rota. Após obter a solução gulosa, o aprimoramento é realizado por trocas aleatórias não repetidas entre a ordem das cidades até conseguir uma rota melhor ou pela quantidade de tentativas definida na Equação 5.

$$\begin{cases} n, & se\ \lfloor n * alpha \rfloor = 0 \\ n * \lfloor n * alpha \rfloor, & se\ \lfloor n * alpha \rfloor > 0 \end{cases} \quad (5)$$

Sendo "*n*" o número de cidades, "*alpha*" um valor entre zero e um $\lfloor n * alpha \rfloor$ arredondamento para baixo de "*n*alpha*". O aprimoramento se repete por "*i*" vezes, sendo este um valor inteiro.

```
PROGRAMA Caixeiro_Viajante_Sequencial

VARIAVEIS
n: Número de cidades;
d: Matriz de distâncias;
s: Solução;
f: Melhor distância encontrada;
t: Tempo de processamento;

s2: Solução temporária local;
f2: Distância encontrada localmente;

i: Número máximo de interações;
alpha: Porcentagem de cidades escolhidas para troca;
v1: Primeiras cidades escolhidas para troca;
v2: Segundas cidades escolhidas para troca;
t_v1: Tamanho total de v1;
t_v2: Tamanho total de v2;

// Seleciona o menor caminho guloso
FUNCAO CONSTRUÇÃO()
    s <- 0;
    f <- valor máximo;

    // Testar caminho guloso iniciando por cada cidade
    PARA i <- 0 ATE n PASSO 1 FACA
        f2 <- 0;
        s2[0] <- i;
        // Método guloso
        PARA j <- 0 ATE n-1 PASSO 1  FACA
            a <- cidade mais próxima não visitada;
            f2 <- f2 + d[s2[j]][a];
            s2[j+1] <- a;
        FIMPARA
        // Verifica o melhor caminho encontrado
        SE f2 < f ENTAO
            s <- s2;
            f <- f2;
        FIMSE
    FIMPARA
```

```
FIMFUNCAO

// Trocar duas cidades aleatoriamente até conseguir uma solução melhor
// ou passar do limite de iterações n*n*alpha
FUNCAO REFINAMENTO_ALEATÓRIO()
    escolha1 <- aleatório(0,n);
    ENQUANTO f2>=f E tamanho(v1) < t_v1 FACA
        SE tamanho(v2) = t_v2 ENTAO
       // Escolher a primeira cidade aleatória não utilizada anteriormente
            escolha1 <- aleatório(0,n);
            ENQUANTO escolha1 C v1 FACA
                escolha1 <- escolha1 + 1;
            FIMENQUANTO
            v2 <- 0;
        FIMSE

        // Escolher a segunda cidade aleatória não utilizada anteriormente
        escolha2 <- aleatório(0,n);
        ENQUANTO escolha2 C v2 OU escolha2 = escolha1 FACA
            escolha2 <- aleatório(0,n);
        FIMENQUANTO
        v1 <- v1 + escolha1;
        v2 <- v2 + escolha2;

        // Trocar as duas cidades e calcular nova rota
        s2 <- troca(s[escolha1], s[escolha2]);
        f2 <- rota(s2);

    FIMENQUANTO

    // Se foi encontrada melhor solução
    SE rota(s2) < rota(s) ENTAO
        s <- s2;
        f <- rota(s);
    FIMSE
FIMFUNCAO

// Função principal
INICIO
    LEIA(d);
    CONSTRUÇÃO();

    i     <- 100;
    alpha <- 0.05;
    f2    <- valor máximo;
    t_v1  <- n;
    t_v2  <- arredonda_para_baixo(n*alpha);
    SE t_v2 = 0 ENTAO
        t_v2 <- 1;
    FIMSE

    PARA i <- 0 ATE i PASSO 1 FACA
        REFINAMENTO_ALEATÓRIO();
    FIMPARA

    ESCREVA(s);
    ESCREVA(f);
    ESCREVA(t);
FIM.
```

Quadro 10 – Caixeiro Viajante sequencial
Fonte: Autoria própria.

A paralelização do algoritmo sequencial foi realizada em duas etapas: a paralelização da fase de construção e da fase de refinamento, pois o método de refinamento contém certa dependência da resposta gulosa. Como o método guloso consiste em verificar as rotas modificando a cidade onde o caixeiro inicia sua trajetória, as "*n*" cidades iniciais foram divididas entre a quantidade de nodos disponíveis entre o *cluster*. Para o método de refinamento foi divida a quantidade de vezes que o método executa ("*i*") entre os nodos, como mostrado no Quadro 11.

```
PROGRAMA Caixeiro_Viajante_Paralelo

VARIAVEIS
n: Número de cidades;
d: Matriz de distâncias;
s: Solução;
f: Melhor distância encontrada;
t: Tempo de processamento;

s2: Solução temporária local;
f2: Distância encontrada localmente;

i: Número máximo de interações;
alpha: Porcentagem de cidades escolhidas para troca;
v1: Primeiras cidades escolhidas para troca;
v2: Segundas cidades escolhidas para troca;
t_v1: Tamanho total de v1;
t v2: Tamanho total de v2;

MESTRE: Nodo mestre do cluster;
NODOS: Conjunto de nodos, exceto o MESTRE;
MPI_ID: ID do processo;
QTD_MPI_ID: Quantidade de processos;
melhor: Nodo que possui melhor reposta;

// Seleciona o menor caminho guloso
FUNCAO CONSTRUÇÃO()
    s <- 0;
    f <- valor máximo;

    // Testar caminho guloso iniciando por cada cidade
    PARA i <- MPI_ID-1 ATE n PASSO i <- i + QTD_MPI_ID FACA
        f2 <- 0;
        s2[0] <- i;
        // Método guloso
        PARA j <- 0 ATE n-1 PASSO 1  FACA
            a <- cidade mais próxima não visitada;
            f2 <- f2 + d[s2[j]][a];
            s2[j+1] <- a;
        FIMPARA
        // Verifica o melhor caminho encontrado
        SE f2 < f ENTAO
            s <- s2;
            f <- f2;
        FIMSE
    FIMPARA
FIMFUNCAO
```

```
// Trocar duas cidades aleatoriamente até conseguir uma solução melhor
// ou passar do limite de iterações n*n*alpha
FUNCAO REFINAMENTO_ALEATÓRIO()
    escolha1 <- aleatório(0,n);
    ENQUANTO f2>=f E tamanho(v1) < t_v1 FACA
        SE tamanho(v2) = t_v2 ENTAO
       // Escolher a primeira cidade aleatória não utilizada anteriormente
            escolha1 <- aleatório(0,n);
            ENQUANTO escolha1 C v1 FACA
                escolha1 <- escolha1 + 1;
            FIMENQUANTO
            v2 <- 0;
        FIMSE

        // Escolher a segunda cidade aleatória não utilizada anteriormente
        escolha2 <- aleatório(0,n);
        ENQUANTO escolha2 C v2 OU escolha2 = escolha1 FACA
            escolha2 <- aleatório(0,n);
        FIMENQUANTO

        v1 <- v1 + escolha1;
        v2 <- v2 + escolha2;

        // Trocar as duas cidades e calcular nova rota
        s2 <- troca(s[escolha1], s[escolha2]);
        f2 <- rota(s2);

    FIMENQUANTO

    // Se foi encontrada melhor solução
    SE rota(s2) < rota(s) ENTAO
        s <- s2;
        f <- rota(s);
    FIMSE
FIMFUNCAO

INICIO

    INICIAR_MPI();
    LEIA(d);

    // Se for o mestre, realizar função de sincronização
    SE MPI_ID = MESTRE ENTAO

        // Receber melhores soluções da fase de construção
        f <- valor máximo;
        PARA i <- 0 ATE QTD_MPI_ID PASSO 1 FACA
            RECEBE_MPI(f2, i+1);
            SE f2 < f ENTAO
                f <- f2;
                melhor <- i+1;
            FIMSE
        FIMPARA

        // Sincroniza melhor solução da construção entre os nodos
        ENVIE_MPI(melhor, NODOS);
        RECEBE_MPI(s, melhor);
        ENVIE_MPI(s, NODOS);

        // Receber melhor solução do Caixeiro Viajante
```

```
        f <- valor máximo;
        PARA i <- 0 ATE QTD_MPI_ID PASSO 1 FACA
            RECEBE_MPI(f2, i+1);
            SE f2 < f ENTAO
                f <- f2;
                melhor <- i+1;
            FIMSE


        FIMPARA

        ENVIE_MPI(melhor, NODOS);
        RECEBE_MPI(s, melhor);

        ESCREVA(s);
        ESCREVA(f);
        ESCREVA(t);

    // Se não for o mestre, realizar função de cálculos
    SENAO

        CONSTRUÇÃO();

        ENVIE_MPI(f, MESTRE); // Enviar solução ao mestre
        RECEBE_MPI(melhor, MESTRE); // Recebe nodo com melhor solução

        // Se for o nodo com melhor solução, enviará a solução
        SE MELHOR = MPI_ID ENTAO
            ENVIE_MPI(s, MESTRE);
            ENVIE MPI(f, NODOS);
            ENVIE_MPI(s, NODOS);
        // Senão, recebera a melhor solução
        SENAO
            RECEBE_MPI(f, melhor);
            RECEBE MPI(s, melhor);
        FIMSE

        i     <- 100;
        alpha <- 0.05;
        f2    <- valor máximo;
        t_v1  <- n;
        t_v2  <- arredonda_para_baixo(n*alpha);
        SE t_v2 = 0 ENTAO
            t_v2 <- 1;
        FIMSE

        PARA i <- QTD_MPI_ID-1 ATE i PASSO i <- i + QTD_MPI_ID FACA
            REFINAMENTO_ALEATÓRIO();
        FIMPARA

        // Enviar melhor local solução ao mestre
        ENVIE_MPI(f, MESTRE);
        RECEBE_MPI(melhor, MESTRE);
        SE MELHOR = MPI_ID ENTAO
            ENVIE_MPI(s, MESTRE);
        FIMSE

    FIMSE
FIM.
```

Quadro 11 – Caixeiro Viajante paralelo
Fonte: Autoria própria.

4.3.3 Sistema de Captura de Dados

A implementação do sistema de captura de dados pode ser divido em duas partes essenciais, o desenvolvimento do código para o microcontrolador Tiva e o desenvolvimento do programa de sincronia, JCentral.

Para o desenvolvimento do código do microcontrolador (APÊNDICE A – Código do microcontrolador Tiva) os seguintes periféricos foram utilizados: TimerA0, TimerA1, UART0 e ADC0. As configurações de cada periférico e suas funções são descritas pelo Quadro 12.

Periférico	**Configuração**	**Função**
TimerA0	Periódico *12.000Hz* ou *83,333us*	Inicializar conversão do ADC
TimerA1	Periódico *60Hz* ou *16,667ms*	Contabilizar um ciclo da rede de alimentação
UART0	115200 BAUD 8 Bits 1 Bit de parada Sem paridade	Comunicação serial com a Central
ADC0	64 amostras simples PD2 como entrada	Conversor analógico do sensor de energia

Quadro 12 – Configuração do microcontrolador
Fonte: Autoria própria.

Com intuito de simplificar o projeto, parte do cálculo de energia é realizado no microcontrolador e parte realizado no JCentral. O microcontrolador fica responsável pela parte da somatória dos quadrados dos valores medidos pelo ADC e dividi-los pela quantidade de amostras. Ao enviar este valor ao JCentral, o mesmo termina o cálculo realizando a operação de raiz quadrada sobre o valor recebido, e multiplicando-o ao valor *110* referente a tensão da rede utilizada. O cálculo foi efetuado desta forma, pois as bibliotecas de codificação para o microcontrolador não possuem métodos matemáticos avançados por definição. A Figura 11 contém o fluxograma do código utilizado no microcontrolador.

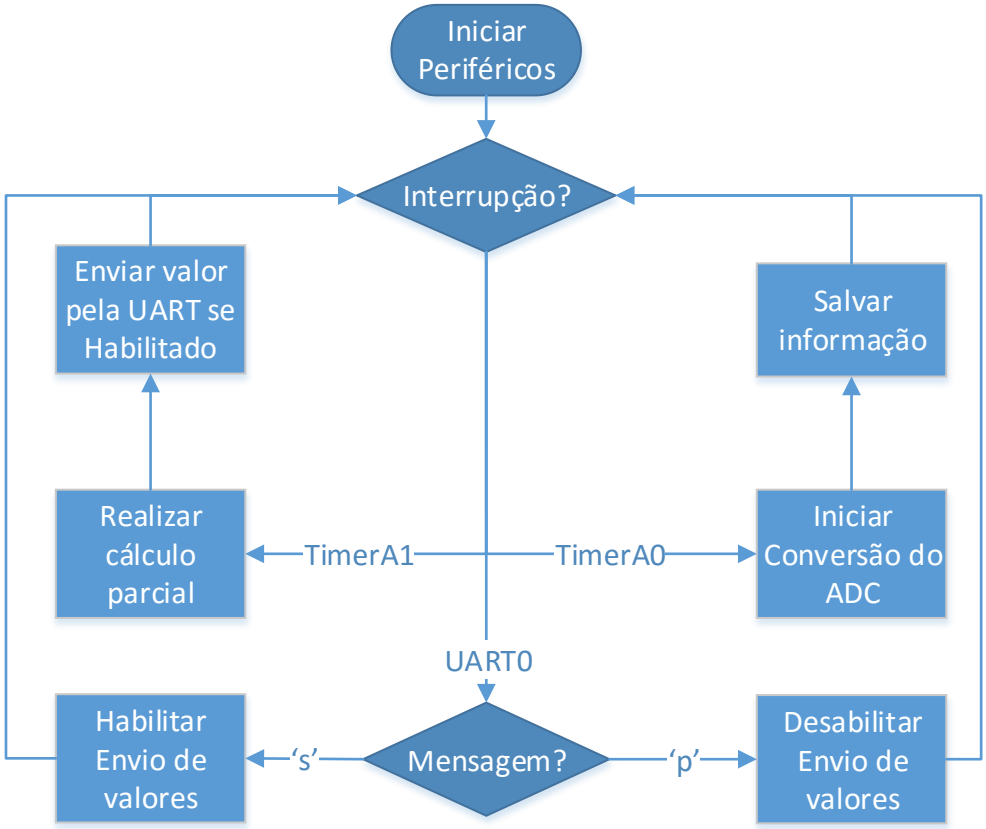

Figura 11 – Fluxograma microcontrolador
Fonte: Autoria própria.

Já para o desenvolvimento do JCentral foram utilizadas quatro classes principais: MainFrame, COMSerial, ReadUART e Shell, descritos na Figura 12. A classe MainFrame é composta basicamente pelas telas de iteração ao usuário, descritas na Seção 4.2 deste trabalho. As classes COMSerial e Shell implementam os métodos de início de comunicação com o microcontrolador e o *cluster* utilizando as bibliotecas citadas anteriormente. Já a classe ReadUART implementa o método de recepção de dados do microcontrolador sincronizado com a execução do algoritmo.

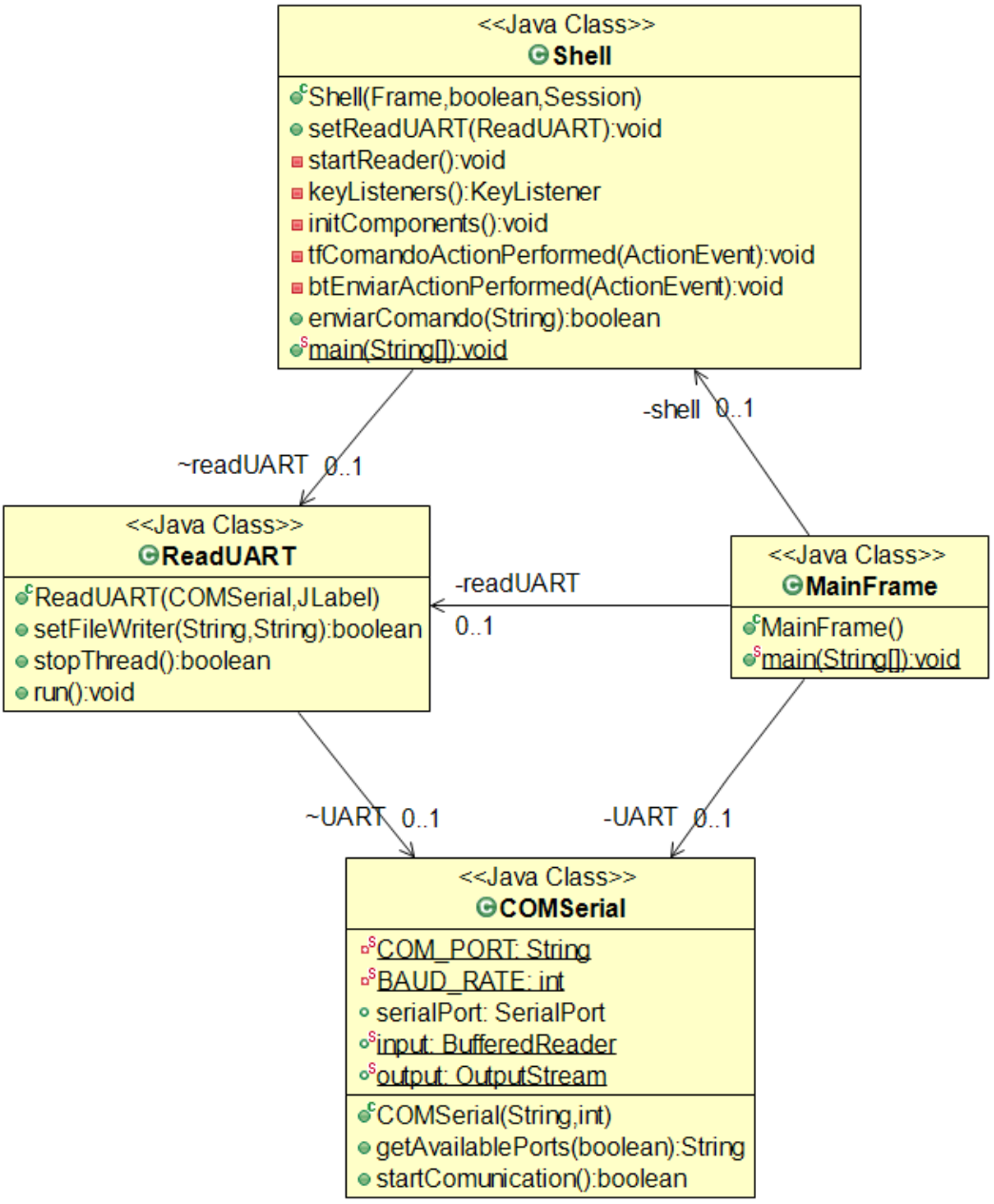

Figura 12 – Diagrama de Classes JCentral
Fonte: Autoria própria.

Para que os dados coletados ocorram de forma síncrona ao *cluster*, é enviado o sinal “s”, definido no desenvolvimento do código, ao microcontrolador para iniciar a captura de energia simultaneamente ao comando de execução do algoritmo. Ao receber o sinal de fim do algoritmo, definida pela mensagem “END”, do *cluster*, a Central envia o sinal de fim de conversão “p” ao microcontrolador. Ao terminar de receber os dados do microcontrolador, os mesmos são calculados e devidamente arquivados. O fluxograma na Figura 13 descreve o funcionamento deste programa.

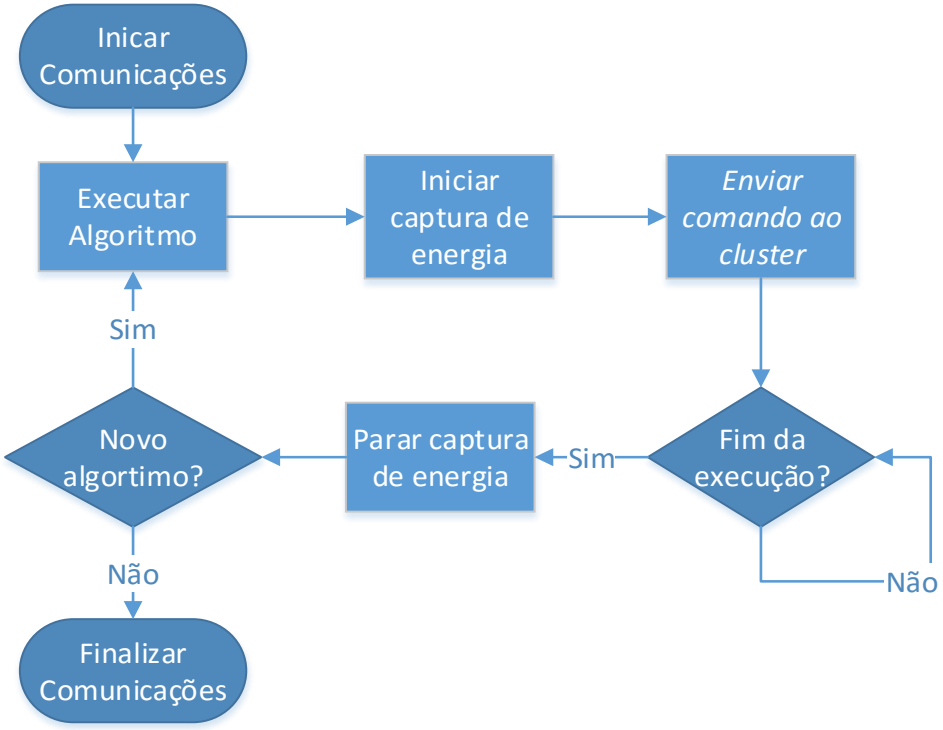

Figura 13 – Fluxograma JCentral
Fonte: Autoria própria.

4.3.4 *Clusters*

Para configurar um *cluster* que utiliza sistemas operacionais baseados em Linux, são necessárias configurações nos arquivos base do Linux e instalação de programas de comunicação entre os dispositivos (PEGUERO, 2015). Para ambos, mestre e escravos, foram necessárias as configurações dos arquivos /etc/hosts (Quadro 13) e /etc/hostname. O arquivo /etc/hosts é composto do endereço IP e nome de cada dispositivo conhecido pela rede, sendo que o nome do dispositivo deve ser único para cada um e previamente alterado no arquivo /etc/hostname. O arquivo /etc/hostname deve conter uma única palavra em uma única linha representando o nome do computador, dessa forma cada dispositivo foi nomeado de acordo com sua função (mestre, nodo1, nodo2, etc.). O Quadro 13 apresenta um exemplo de lista de dispositivos utilizados nos *clusters*, sendo que "*X*" é o último nodo disponível.

127.0.0.1	localhost
192.168.0.100	master
192.168.0.101	node1
192.168.0.102	node2
192.168.0.X	nodeX

Quadro 13 – Configuração arquivo /etc/hosts
Fonte: Autoria própria.

A configuração do endereço IP de cada dispositivo pode ser realizada de forma dinâmica, utilizando um servidor DHCP, ou de forma estática, alterando o arquivo de configuração /etc/network/interfaces do Linux. Pela quantidade de nodos utilizados neste trabalho e futura simplicidade da execução dos programas, foi utilizada a metodologia por endereço estático, como segue o Quadro 14.

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
address 192.168.0.100
netmask 255.255.255.0
network 192.168.0.0
broadcast 192.168.0.255
```

Quadro 14 – Configuração /etc/network/interfaces
Fonte: Autoria própria.

Para cada dispositivo, o valor de "*address*" deve ser único, assim como na lista de configuração do arquivo "/etc/hosts". Após definidos os nomes e endereços, foi configurado um usuário padrão em cada um dos nodos de cada *cluster*, utilizando os seguintes comandos:

1. "sudo adduser mpiuser --uid 999"
2. "sudo usermod –a –G sudo mpiuser"
3. "sudo passwd mpiuser mpiuser"

Desta forma é criado o usuário “mpiuser” com a senha “mpiuser” com privilégios de administrador. Para finalizar é instalada a biblioteca MPICH2 em cada dispositivo, utilizando o seguinte comando “sudo apt-get install mpich2”.

Para o nodo mestre foi necessário instalar e configurar as versões servidores dos programas NFS e SSH. Segue os passos para instalação e configuração do NFS Server:

1. “sudo apt-get install nfs-kernel-server”;
2. “echo “/user/mpiuser *(rw, sync ,no_subtree_check)” >> /etc/exports”;
3. “sudo service nfs-kernel-server restart”.
4. “sudo exportfs –a”

Desta forma a pasta “/user/mpiuser” pertinente ao usuário “mpiuser” é compartilhado com qualquer computador com acesso. Para instalar e configurar o SSH Server, de forma a conectar em outros nodos sem requisição de senha, foram seguidos os seguintes passos:

1. “sudo apt-get install openssh-server”;
2. “su - mpiuser”;
3. “ssh-keygen –t –rsa”;
4. “cd /user/mpiuser/.ssh”;
5. “cat id_pub.dsa >> authorized_keys”.

Configurado o nodo mestre, segue-se com as configurações dos nodos escravos, para tal foram utilizados os seguintes passos:

1. “sudo apt-get install nfs-common”;
2. “sudo echo “master:/home/user/mpiuser /home/mpiuser nfs” >> /etc/fstab”;
3. “sudo reboot now”.

Esses comandos instalam a versão cliente do programa NFS no escravo e o configura com a pasta compartilhada pelo nodo mestre. Com isso realizado o *cluster* está devidamente configurado para uso.

5 DADOS OBTIDOS E DISCUSSÕES

Os dados coletados dos *clusters Desktop,* Raspberry e Cubietruck estão dispostos nas tabelas de custo de construção do *cluster*, HP Linpack, tempo, *speedup* e medidas de energia, sendo que a primeira coluna de cada tabela é composta da quantidade de nodos utilizados em cada problema. A linha que contém zero nodo é referente à execução dos algoritmos sequenciais, visto que a topologia utilizada para paralelização dos algoritmos é de mestre/escravo, sendo necessário ao menos dois nodos.

Para o *cluster* de Cubietruck não há tabelas de medidas de energia, pois foram utilizadas portas USB para alimentá-lo, de forma que não foi possível utilizar o dispositivo para mensurá-lo. Apesar de não ter sido mensurado a energia gasta pelo *cluster* Cubietruck com o dispositivo, é possível estimar o custo, visto que a fonte utilizada pelo Cubietruck pode ser a mesma utilizada pelo Raspberry (5W), e que a potência utilizada quando o dispositivo está ou não trabalhando é aproximadamente a mesma.

5.1 CUSTOS DE CONSTRUÇÃO DOS CLUSTERS

Com os preços de cada dispositivo utilizado para criação dos *clusters* dispostos na Seção 3, e utilizando a Equação 6 para cálculo de custo, obteve-se a Tabela 2 e o gráfico da Figura 14.

$$C = Q * (D + F) \tag{6}$$

Sendo:

C – Custo final;

Q – Quantidade de dispositivos;

D – Dispositivo utilizado (*desktop,* Raspberry ou Cubietruck);

F – Fonte utilizada.

Tabela 2 – Custo em R$ de cada tipo de *cluster*

Dispositivos	Desktop	Raspberry	Cubietruck
1	R$ 1.498,00	R$ 313,81	R$ 531,01
2	R$ 2.996,00	R$ 627,62	R$ 1.062,02
3	R$ 4.494,00	R$ 941,43	R$ 1.593,03
4	R$ 5.992,00	R$ 1.255,24	R$ 2.124,04
5	R$ 7.490,00	R$ 1.569,05	R$ 2.655,05
6	R$ 8.988,00	R$ 1.882,86	R$ 3.186,06
7	R$ 10.486,00	R$ 2.196,67	R$ 3.717,07
8	R$ 11.984,00	R$ 2.510,48	R$ 4.248,08
9	R$ 13.482,00	R$ 2.824,29	R$ 4.779,09
10	R$ 14.980,00	R$ 3.138,10	R$ 5.310,10
20	R$ 29.960,00	R$ 6.276,20	R$ 10.620,20
30	R$ 44.940,00	R$ 9.414,30	R$ 15.930,30
40	R$ 59.920,00	R$ 12.552,40	R$ 21.240,40
50	R$ 74.900,00	R$ 15.690,50	R$ 26.550,50
60	R$ 89.880,00	R$ 18.828,60	R$ 31.860,60
70	R$ 104.860,00	R$ 21.966,70	R$ 37.170,70
80	R$ 119.840,00	R$ 25.104,80	R$ 42.480,80
90	R$ 134.820,00	R$ 28.242,90	R$ 47.790,90
100	R$ 149.800,00	R$ 31.381,00	R$ 53.101,00

Fonte: Autoria própria.

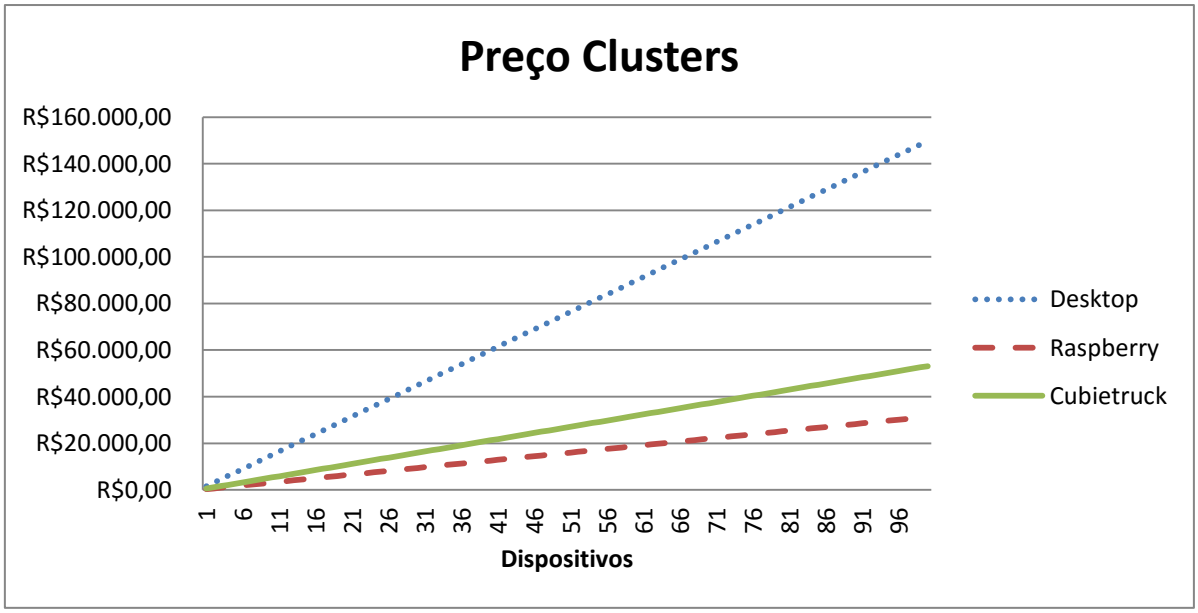

Figura 14 – Gráfico do custo em R$ de cada tipo de *cluster*
Fonte: Autoria própria.

5.2 RESULTADOS HP LINPACK

As Tabelas 3, 4 e 5 e as Figuras 15 e 16 apresentam os resultados em Gflops provenientes das execuções do benchmark HP Linpack nos clusters, assim como as configurações utilizadas e o tempo de cálculo necessário. A coluna “N” representa o número de problemas executados, a coluna “NB” é o número do bloco dos problemas e as colunas “P” e “Q” são referentes aos números de linhas e colunas que o processo executará.

Tabela 3 – Resultados HP Linpack do *cluster desktop*

Nodos	N	NB	P	Q	Tempo (s)	Gflops
0	20352	192	1	2	324,99	1,729E+01
1	29184	192	2	2	539,34	3,073E+01
2	35712	192	2	3	692,75	4,383E+01
3	41088	192	2	4	799,52	5,784E+01
4	46080	192	2	5	914,28	7,135E+01
5	50688	192	3	4	1050,93	8,262E+01
6	54528	192	2	7	1103,80	9,793E+01

Fonte: Autoria própria.

Tabela 4 – Resultados HP Linpack do *cluster* Raspberry

Nodos	N	NB	P	Q	Tempo (s)	Gflops
0	7000	168	1	1	1276,64	1,792E-01
1	10000	168	1	2	1953,68	3,413E-01
2	12000	168	1	3	2239,19	5,146E-01
3	14000	168	2	2	2781,03	6,579E-01
4	16000	168	1	5	3957,30	6,901E-01
5	17000	168	2	3	3353,16	9,769E-01
6	18000	168	1	7	3550,37	1,095E+00
7	20496	168	2	4	4556,99	1,170E+00

Fonte: Autoria própria.

Tabela 5 – Resultados HP Linpack do *cluster* Cubietruck

Nodos	N	NB	P	Q	Tempo (s)	Gflops
0	13056	192	1	2	2713,16	5,469E-01
1	18432	192	2	2	3952,95	1,056E+00
2	22656	192	2	3	5074,14	1,528E+00
3	26112	192	2	4	5912,10	2,008E+00
4	29184	192	2	5	6661,42	2,488E+00
5	32064	192	3	4	7452,23	2,949E+00
6	24560	192	2	7	7910,92	3,479E+00
7	37056	192	4	4	8614,02	3,938E+00

Fonte: Autoria própria.

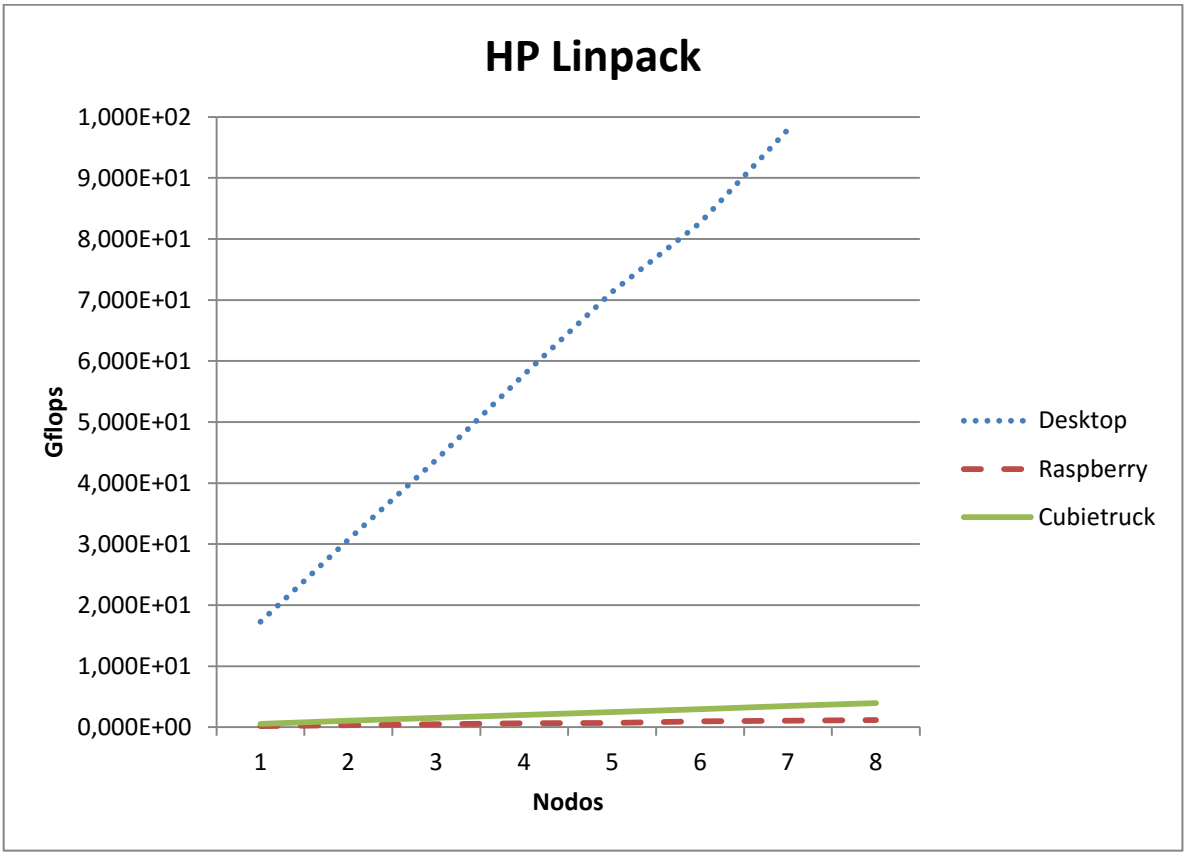

Figura 15 – Gráfico, todos os resultados HP Linpack
Fonte: Autoria própria.

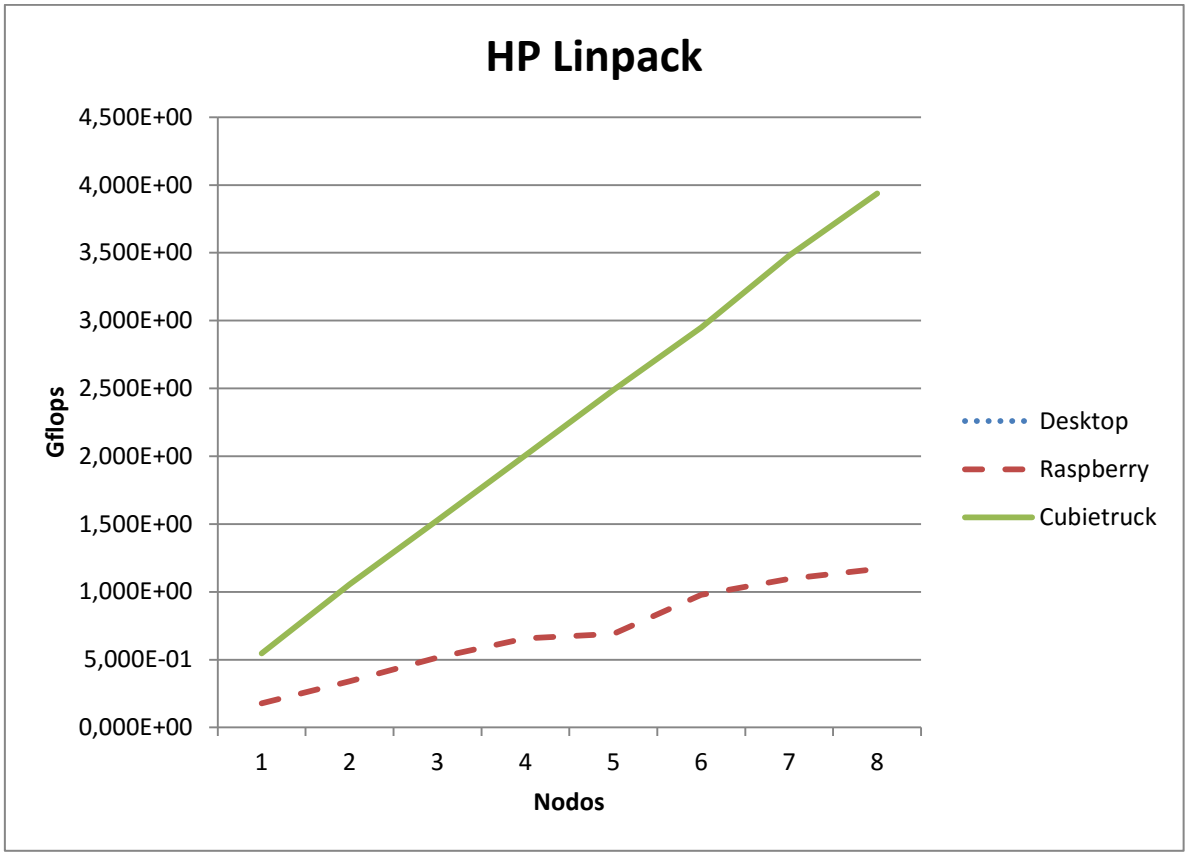

Figura 16 – Gráfico, resultados HP Linpack Raspberry e Cubietruck
Fonte: Autoria própria.

Observando-se os resultados provenientes da ferramenta HP Linpack, pode-se constatar a diferença de processamento entre os *clusters*, sendo que o *cluster desktop* pode chegar a realizar cem vezes mais operações que o Raspberry e trinta vezes mais operações que o Cubietruck.

5.3 RESULTADOS DE TEMPO, SPEEDUP E ENERGIA

A seguir foram realizadas coletas de dados dos *clusters* (tempo, *speedup* e energia) utilizando os algoritmos implementados anteriormente (Multiplicação de Matriz e Caixeiro Viajante). Para cada entrada de dados foram coletadas seis amostras de tempo e energia, as tabelas a seguir mostram os dados em formato de média aritmética das amostras nas colunas de acordo com a entrada de dados utilizados pelos programas. Como o algoritmo do Caixeiro Viajante utiliza arquivos provenientes da TSPLIB, os nomes deles foram utilizados para descrever suas entradas, sendo que as letras identificam o autor dos dados e os números subsequentes são relativos à quantidade de nós do problema. Para o programa de multiplicação de matriz, foram utilizadas matrizes aleatórias de “*m*” linhas por “*n*” colunas e “*n*” linhas por “*p*” colunas, respectivamente. Para diminuir o nome da entrada, foi utilizada a nomenclatura “*m x n x p*”.

5.3.1 Resultados de tempo e *speedup*

As Tabelas 6, 7 e 8 são referentes aos tempos em segundos de execução para o problema de Multiplicação de Matriz para os *clusters desktop*, Raspberry e Cubietruck, respectivamente.

Tabela 6 – Tempo em segundos (*s*) da Multiplicação de Matriz no *cluster desktop*

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	3,653	5,234	21,202	49,219	91,687
1	1,519	1,489	2,891	6,394	11,120
2	1,332	1,317	1,912	3,840	6,418
3	1,033	1,064	1,745	4,154	6,320
4	1,020	1,044	1,477	2,619	3,960
5	0,895	0,952	1,411	2,341	3,550
6	1,015	1,052	1,439	2,852	3,443

Fonte: Autoria própria.

Tabela 7 – Tempo em segundos (*s*) da Multiplicação de Matriz no *cluster* Raspberry

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	118,334	164,573	487,294	1076,756	1952,776
1	114,847	161,013	489,139	1156,382	2142,216
2	60,349	83,205	252,249	572,437	1091,915
3	42,253	57,676	171,641	394,527	735,490
4	34,766	46,215	131,544	311,896	571,764
5	28,557	38,205	108,591	254,291	468,627
6	25,432	33,537	95,131	223,114	399,374
7	23,298	30,458	81,974	181,989	331,900

Fonte: Autoria própria.

Tabela 8 – Tempo em segundos (*s*) da Multiplicação de Matriz no *cluster* Cubietruck

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	44,230	60,436	178,955	408,420	710,531
1	23,867	32,282	96,170	279,926	499,040
2	14,115	18,455	52,131	147,702	222,009
3	10,333	13,891	37,448	103,103	176,692
4	12,213	11,472	29,914	81,059	135,520
5	8,253	10,344	25,770	67,059	112,185
6	7,634	10,226	22,393	58,418	95,461
7	7,217	9,618	21,451	54,873	85,774

Fonte: Autoria própria.

A Figura 17 apresenta os dados das Tabelas 6, 7 e 8 de forma gráfica para cada tipo de entrada para o problema de Multiplicação de Matriz.

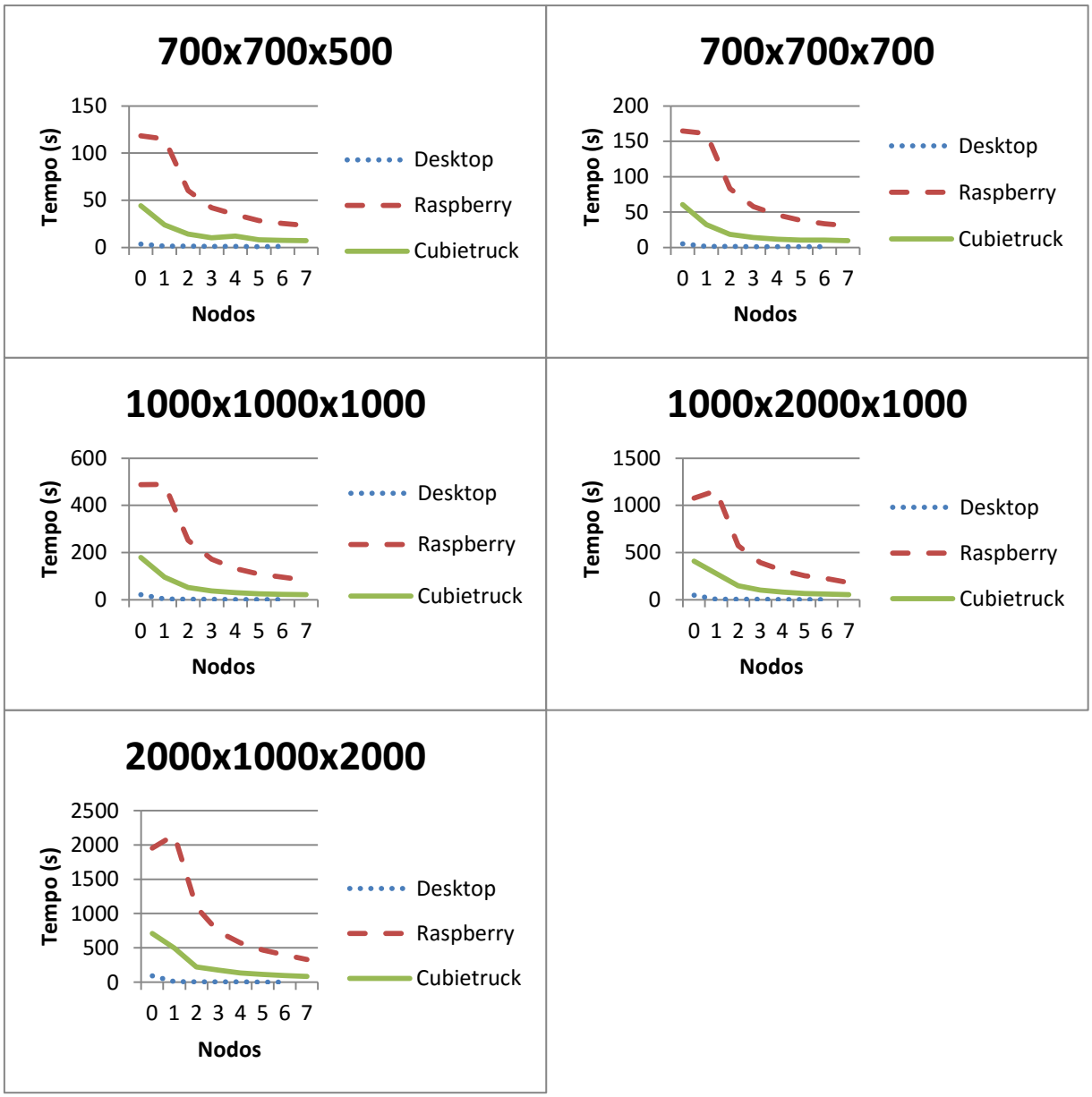

Figura 17 – Gráficos de tempo para o problema de Multiplicação de Matriz
Fonte: Autoria própria.

As Tabelas 9, 10 e 11 são referentes ao *speedup* para o problema de Multiplicação de Matriz para os *clusters desktop*, Raspberry e Cubietruck, respectivamente. Como não há mudanças no *speedup* para 0 nodos, a linha foi omitida.

Tabela 9 – *Speedup* da Multiplicação de Matriz no *cluster desktop*

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
1	2,405	3,515	7,334	7,698	8,245
2	2,742	3,974	11,089	12,817	14,286
3	3,536	4,919	12,150	11,849	14,507
4	3,581	5,013	14,355	18,793	23,153
5	4,082	5,498	15,026	21,025	25,827
6	3,599	4,975	14,734	17,258	26,630

Fonte: Autoria própria.

Tabela 10 – *Speedup* da Multiplicação de Matriz no *cluster* Raspberry

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
1	1,030	1,022	0,996	0,931	0,912
2	1,961	1,978	1,932	1,881	1,788
3	2,801	2,853	2,839	2,729	2,655
4	3,404	3,561	3,704	3,452	3,415
5	4,144	4,308	4,487	4,234	4,167
6	4,653	4,907	5,122	4,826	4,890
7	5,079	5,403	5,944	5,917	5,884

Fonte: Autoria própria.

Tabela 11 – *Speedup* da Multiplicação de Matriz no *cluster* Cubietruck

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
1	1,853	1,872	1,861	1,459	1,424
2	3,134	3,275	3,433	2,765	3,200
3	4,280	4,351	4,779	3,961	4,021
4	3,622	5,268	5,982	5,039	5,243
5	5,359	5,843	6,944	6,090	6,334
6	5,794	5,910	7,992	6,991	7,443
7	6,129	6,284	8,343	7,443	8,284

Fonte: Autoria própria.

A Figura 18 apresenta os dados das Tabelas 9, 10 e 11 de forma gráfica para cada tipo de entrada para o problema de Multiplicação de Matriz.

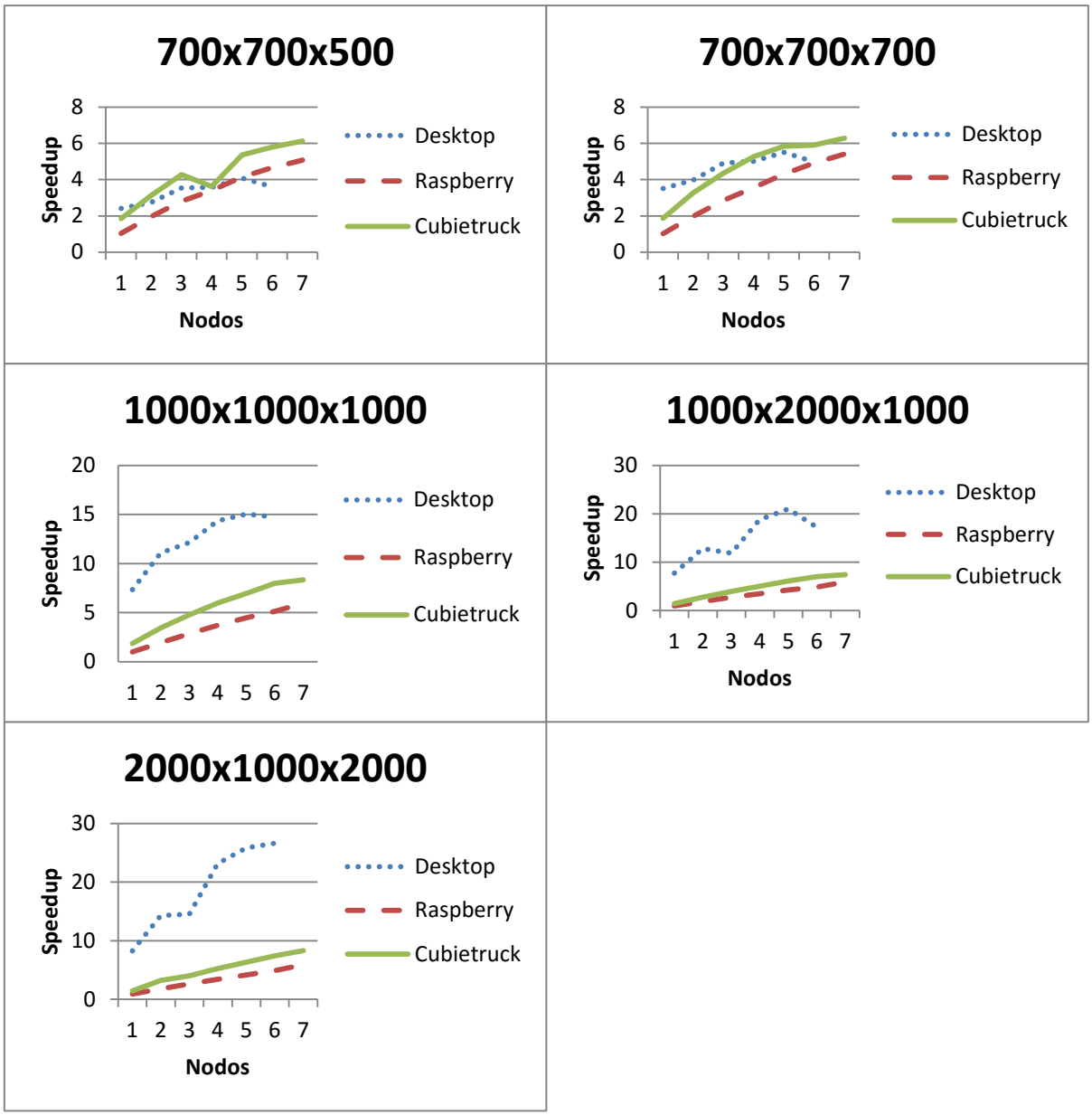

Figura 18 – Gráficos de *speedup* para o problema de Multiplicação de Matriz
Fonte: Autoria própria.

As Tabelas 12, 13 e 14 são referentes aos tempos em segundos de execução para o problema do Caixeiro Viajante para os *clusters desktop*, Raspberry e Cubietruck, respectivamente.

Tabela 12 – Tempo em segundos (*s*) do Caixeiro Viajante no *cluster desktop*

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
0	0,191	0,355	3,313	12,023	19,660	143,326	195,320	657,029	6139,487
1	0,647	0,707	2,305	6,871	10,833	75,049	102,319	344,015	3255,733
2	0,637	0,703	1,658	3,753	5,662	37,731	51,240	173,025	1629,421
3	0,588	0,558	1,194	2,723	3,938	25,800	34,704	116,046	1088,446
4	0,615	0,622	1,066	2,218	3,241	19,692	26,315	86,771	819,080
5	0,605	0,638	0,922	1,844	2,594	16,371	21,456	70,455	654,764
6	0,727	0,670	0,914	1,712	2,368	13,893	17,808	59,579	546,542

Fonte: Autoria própria.

Tabela 13 – Tempo em segundos (*s*) do Caixeiro Viajante no *cluster* Raspberry

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
0	0,908	2,164	25,639	91,916	152,158	1105,890	1524,234	4942,692	45018,361
1	1,814	2,834	26,393	92,187	152,090	1109,664	1530,830	4905,565	44693,805
2	1,896	2,274	13,570	46,854	75,952	525,880	727,637	2397,894	22218,599
3	1,803	2,197	9,521	31,745	50,203	350,225	480,901	1589,294	14816,749
4	1,956	2,171	7,701	24,475	38,756	259,560	354,226	1184,424	11135,330
5	2,167	2,371	6,852	20,274	31,438	214,662	290,661	952,456	8906,623
6	2,379	2,534	6,355	17,546	26,897	177,855	242,755	807,566	7432,747
7	2,861	2,961	6,059	15,614	23,817	154,496	209,998	689,050	6389,307

Fonte: Autoria própria.

Tabela 14 – Tempo em segundos (*s*) do Caixeiro Viajante no *cluster* Cubietruck

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
0	0,576	1,608	15,553	53,393	94,835	631,983	862,428	2903,156	27510,882
1	0,785	1,511	8,509	27,479	44,099	309,386	429,477	1428,913	13440,633
2	0,894	1,223	4,897	14,607	22,674	155,240	212,799	713,548	6759,207
3	0,934	1,249	4,170	10,401	15,918	104,387	142,578	475,986	4504,201
4	0,945	1,178	4,052	8,411	12,423	79,769	106,633	358,397	3398,497
5	1,077	1,167	3,124	7,376	10,396	64,093	87,662	294,904	2707,063
6	1,007	1,177	2,787	6,328	8,895	53,323	72,586	242,520	2265,139
7	1,064	1,291	3,443	5,933	8,506	46,032	62,792	206,881	1941,367

Fonte: Autoria própria.

A Figura 19 apresenta os dados das Tabelas 12, 13 e 14 de forma gráfica para cada tipo de entrada para o problema do Caixeiro Viajante.

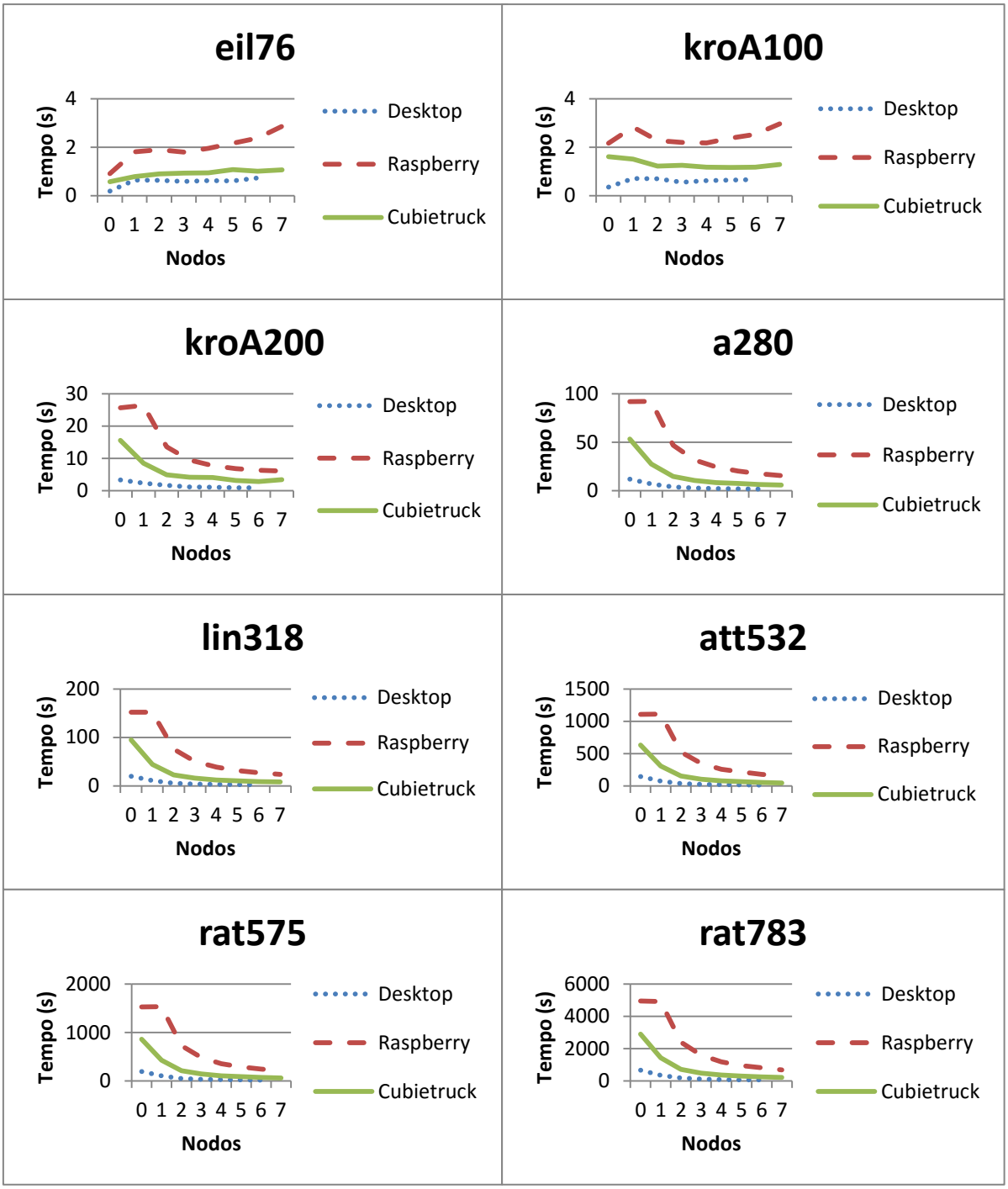

Figura 19 – Gráficos de tempo para o problema do Caixeiro Viajante
Fonte: Autoria própria.

As Tabelas 15, 16 e 17 são referentes ao *speedup* para o problema do Caixeiro Viajante para os *clusters desktop*, Raspberry e Cubietruck, respectivamente. Como não há mudanças no *speedup* para 0 nodos, a linha foi omitida.

Tabela 15 – *Speedup* do Caixeiro Viajante no *cluster desktop*

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
1	0,295	0,502	1,437	1,750	1,815	1,910	1,909	1,910	1,886
2	0,300	0,505	1,998	3,204	3,472	3,799	3,812	3,797	3,768
3	0,325	0,636	2,775	4,415	4,992	5,555	5,628	5,662	5,641
4	0,311	0,571	3,108	5,421	6,066	7,278	7,422	7,572	7,496
5	0,316	0,556	3,593	6,520	7,579	8,755	9,103	9,326	9,377
6	0,263	0,530	3,625	7,023	8,302	10,316	10,968	11,028	11,233

Fonte: Autoria própria.

Tabela 16 – *Speedup* do Caixeiro Viajante no *cluster* Raspberry

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
1	0,501	0,764	0,971	0,997	1,000	0,997	0,996	1,008	1,007
2	0,479	0,952	1,889	1,962	2,003	2,103	2,095	2,061	2,026
3	0,504	0,985	2,693	2,895	3,031	3,158	3,170	3,110	3,038
4	0,464	0,997	3,329	3,756	3,926	4,261	4,303	4,173	4,043
5	0,419	0,913	3,742	4,534	4,840	5,152	5,244	5,189	5,054
6	0,382	0,854	4,034	5,239	5,657	6,218	6,279	6,120	6,057
7	0,317	0,731	4,232	5,887	6,389	7,158	7,258	7,173	7,046

Fonte: Autoria própria.

Tabela 17 – *Speedup* do Caixeiro Viajante no *cluster* Cubietruck

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783	nrw1379
1	0,734	1,064	1,828	1,943	2,151	2,043	2,008	2,032	2,047
2	0,644	1,315	3,176	3,655	4,183	4,071	4,053	4,069	4,070
3	0,617	1,287	3,730	5,133	5,958	6,054	6,049	6,099	6,108
4	0,610	1,365	3,838	6,348	7,634	7,923	8,088	8,100	8,095
5	0,535	1,378	4,979	7,239	9,122	9,860	9,838	9,844	10,163
6	0,572	1,366	5,581	8,438	10,662	11,852	11,881	11,971	12,145
7	0,541	1,246	4,517	8,999	11,149	13,729	13,735	14,033	14,171

Fonte: Autoria própria.

A Figura 20 apresenta os dados das Tabelas 15, 16 e 17 de forma gráfica para cada tipo de entrada para o problema do Caixeiro Viajante.

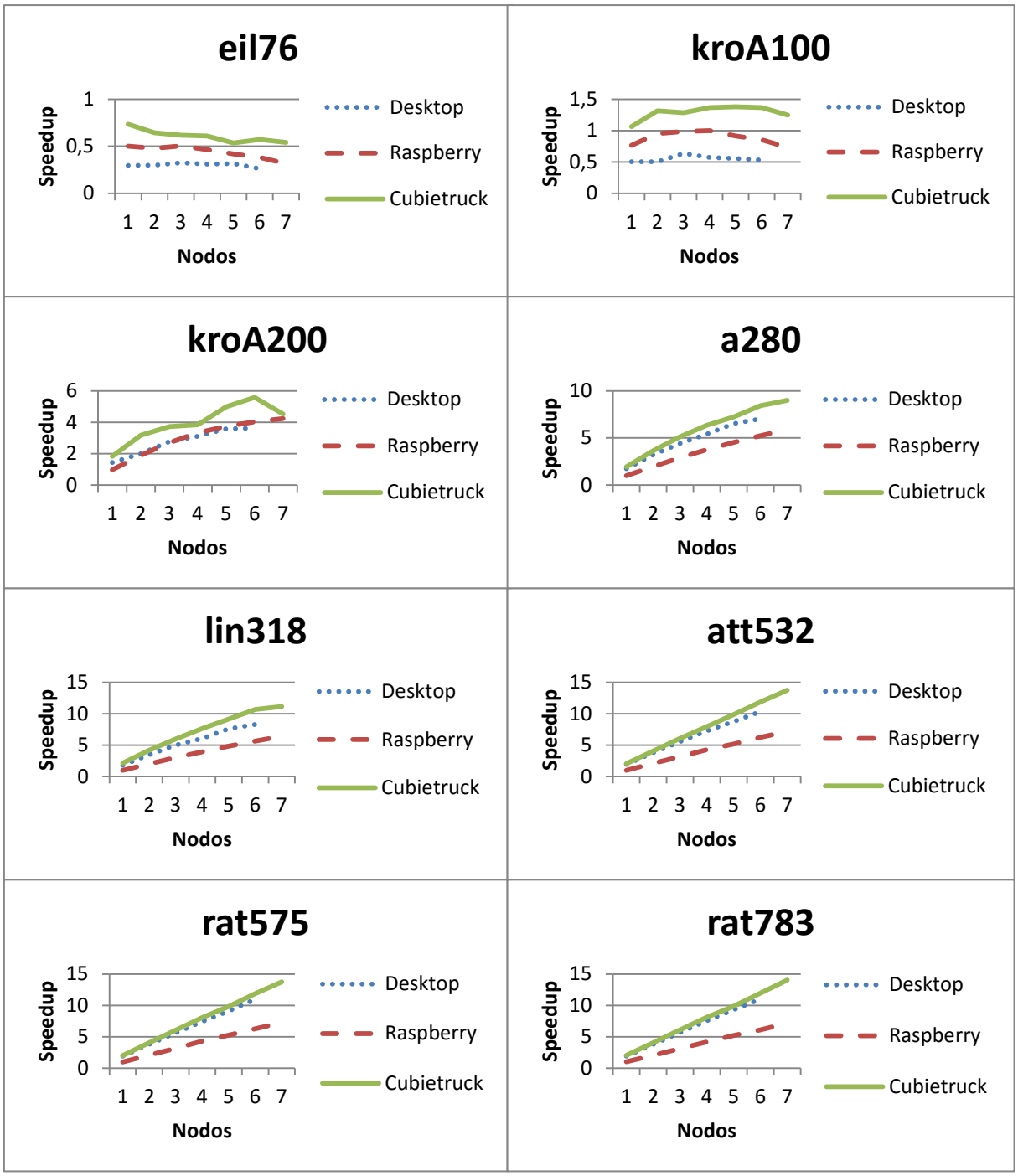

Figura 20 – Gráficos de *speedup* para o problema do Caixeiro Viajante
Fonte: Autoria própria.

5.3.2 Resultados custo energético

Analisando os dados nas tabelas e os gráficos de *speedup*, é possível notar que *clusters* possuem baixos ganhos de *speedup* utilizando entradas com baixa necessidade de processamento, tanto para o problema de Multiplicação de Matriz, quanto para o problema do Caixeiro Viajante. Isso impactará diretamente em um custo maior de energia ao adicionar mais nodos ao *cluster*, o que é mostrado a seguir com os dados de energia consumida.

Apesar dos *clusters* Raspberry e Cubietruck apresentarem melhores *speedup* em alguns casos, não se deve associar isso diretamente ao tempo de processamento, pois ao comparar com os dados de tempo nota-se a diferença entre o tempo de execução do *cluster desktop* em relação aos outros.

5.3.2 Resultados de consumo energético

Antes de obter os dados de energia consumida dos *clusters*, foram realizadas seis coletas de amostras de energia antes e durante a execução dos algoritmos, resultando a Tabela 18.

Tabela 18 – Potência média em *watts* dos *clusters* em repouso e trabalhando

Nodos	Raspberry Pi (Repouso)	Raspberry Pi (Trabalhando)	*Desktop* HP (Repouso)	*Desktop* HP (Trabalhando)
0	5,007	6,579	49,538	87,480
1	10,976	11,928	96,278	191,606
2	15,586	16,498	141,725	277,512
3	20,774	21,543	191,125	363,303
4	25,702	27,088	253,223	487,125
5	29,750	31,672	295,984	570,580
6	33,877	37,007	375,504	666,453
7	40,684	41,694	-	-

Fonte: Autoria própria.
*Como foram utilizados sete computadores *desktop* no total, não há dados para a sétima linha, assim como descrito no capítulo “Materiais e Métodos”.

Com isso é possível verificar que, apesar dos computadores *desktop* utilizarem maiores quantidades de energia ao processar, eles também possuem maior controle sobre o nível de energia gasta.

As Tabelas 19, 20 e 21 são referentes ao consumo energético em watt segundo do problema de Multiplicação de Matriz para os *clusters desktop*, Raspberry e Cubietruck, respectivamente.

Tabela 19 – Custo energético (*Ws*) da Multiplicação de Matriz no *cluster desktop*

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	336,297	475,861	1944,480	4390,430	8284,190
1	280,079	272,767	580,202	1278,310	2280,245
2	326,565	358,979	558,292	1148,419	1949,200
3	361,044	378,153	652,045	1437,462	2243,377
4	472,197	482,096	749,869	1434,694	2206,399
5	484,434	538,601	835,300	1452,752	2241,947
6	632,911	678,090	1001,029	2104,575	2578,290

Fonte: Autoria própria.

Tabela 20 – Custo energético (*Ws*) da Multiplicação de Matriz no *cluster* Raspberry

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	778,125	1089,185	3204,008	7106,385	12863,859
1	1436,955	2064,319	6216,467	12804,660	25875,117
2	996,536	1372,122	4165,184	9534,342	18088,822
3	899,174	1228,933	3662,961	8530,042	16133,444
4	930,918	1240,513	3530,655	8693,044	15863,834
5	920,019	1229,082	3459,342	8071,287	14682,489
6	949,801	1285,017	3619,871	8177,015	14464,359
7	987,969	1303,836	3532,580	7521,049	13688,657

Fonte: Autoria própria.

Tabela 21 – Custo energético (*Ws*) da Multiplicação de Matriz no *cluster* Cubietruck

Nodos	700x700x500	700x700x700	1000x1000x1000	1000x2000x1000	2000x1000x2000
0	221,150	302,180	894,775	2042,100	3552,655
1	238,670	322,820	961,700	2799,260	4990,400
2	211,725	276,825	781,965	2215,530	3330,135
3	206,660	277,820	748,960	2062,060	3533,840
4	305,325	286,800	747,850	2026,475	3388,000
5	247,590	310,320	773,100	2011,770	3365,550
6	267,190	357,910	783,755	2044,630	3341,135
7	288,680	384,720	858,040	2194,920	3430,960

Fonte: Autoria própria.

A Figura 21 apresenta os dados das Tabelas 19, 20 e 21 de forma gráfica para cada tipo de entrada para o problema de Multiplicação de Matriz.

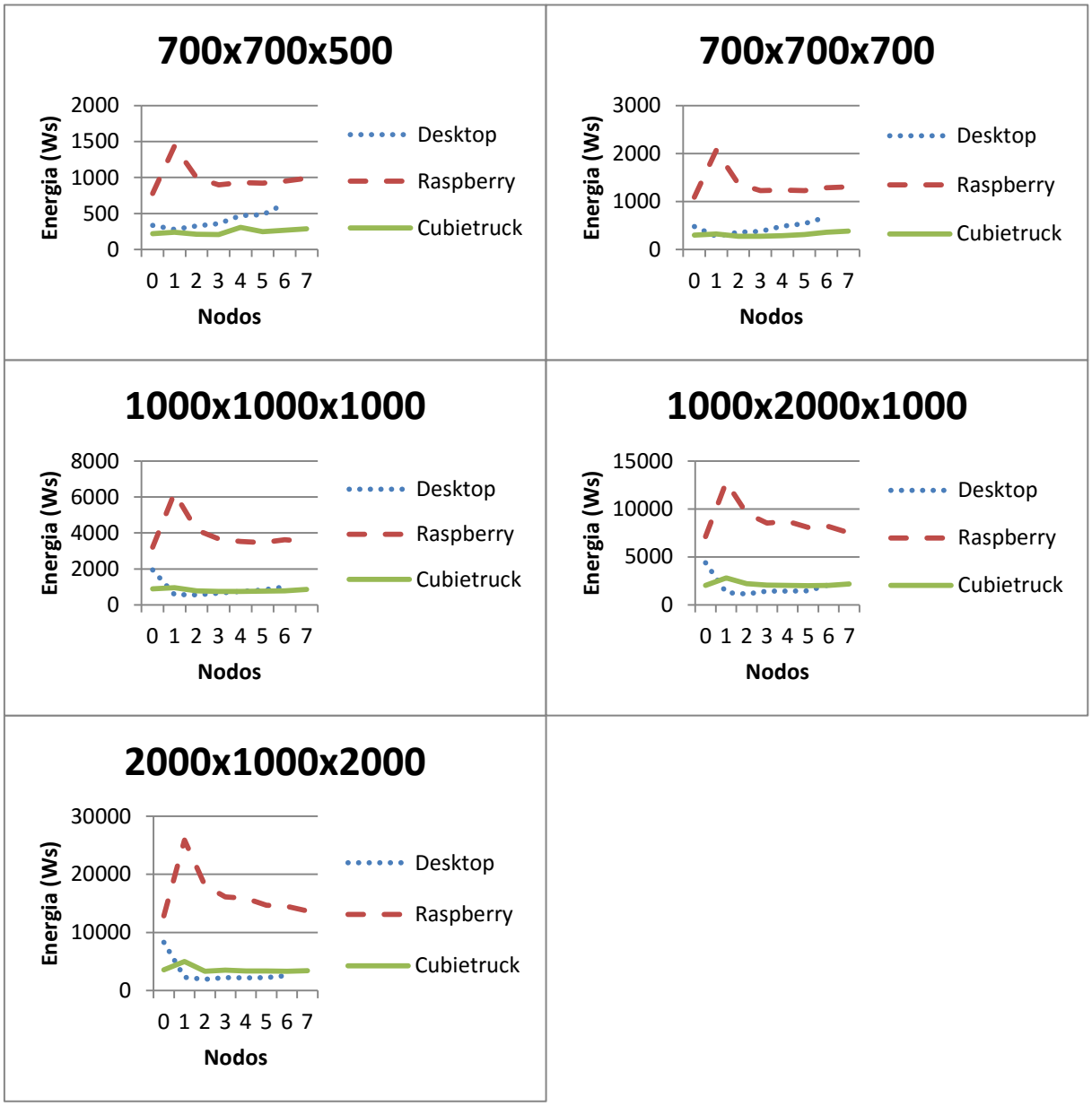

Figura 21 – Resultados custo energético para Multiplicação de Matriz
Fonte: Autoria própria.

Apesar dos dispositivos Raspberry utilizarem menos potência para mantê-los ativos, nota-se que para resolução dos problemas de Multiplicação de Matriz ele possui o maior consumo energético entre os *clusters*, condição essa que se dá ao fato do seu tempo de processamento ser elevado. Além disso, é possível notar um pico de energia quando o *cluster* Raspberry passa de zero nodo para um nodo, condição essa dada pelo modo da arquitetura utilizada para realizar os cálculos. Como o dispositivo Raspberry não possuí múltiplos núcleos, todo o trabalho executado anteriormente unicamente pelo mestre, agora é passado somente ao único núcleo do escravo, resultando no mesmo tempo de processamento com o dobro de energia consumida.

Entretanto o *cluster* Cubietruck, utilizando a mesma potência de um Raspberry, apresenta um desempenho melhor com a energia, sendo melhor que o *cluster desktop* para entradas de problemas menores. Como os dispositivos Cubietruck possuem múltiplos núcleos, os resultados dos testes não possuem a mesma discrepância quando o *cluster* passa de zero nodo para um nodo.

Analisando o comportamento do consumo de energia para as entradas de dados, pode-se supor que quanto maior a necessidade de processamento, melhor será o desempenho do *cluster desktop*, resultando em menor custo energético em relação aos outros *clusters*. Cruzando os dados de energia com os dados de *speedup*, é possível notar que quanto maior o *speedup* do *cluster*, maior a tendência dele consumir menos energia, assim como o *cluster desktop*.

As Tabelas 22, 23 e 24 são referentes ao consumo energético em watts segundo do problema do Caixeiro Viajante para os *clusters desktop*, Raspberry e Cubietruck, respectivamente.

Tabela 22 – Custo energético (*Ws*) do Caixeiro Viajante no *cluster desktop*

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783
0	12,764	25,512	294,061	1083,528	1538,260	13034,120	18115,515	61355,177
1	88,398	106,144	427,550	1351,960	2208,331	15479,585	21679,809	74823,779
2	135,940	145,105	395,147	1083,683	1675,662	11469,022	15931,007	55793,192
3	173,266	168,334	411,820	1040,410	1558,269	10599,362	14217,610	49430,755
4	228,819	234,131	443,932	1067,470	1571,073	10182,572	13995,410	48097,189
5	272,566	257,465	469,254	1041,138	1524,108	9977,495	13449,865	46217,545
6	347,797	352,212	538,789	1117,950	1618,875	9967,650	13083,474	45278,970

Fonte: Autoria própria.

Tabela 23 – Custo energético (*Ws*) do Caixeiro Viajante no *cluster* Raspberry

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783
0	5,836	14,045	169,948	609,860	1009,487	7281,707	10036,510	32331,695
1	21,029	35,005	320,779	1096,441	1721,158	12199,752	16839,644	57114,952
2	30,431	36,120	224,523	775,650	1254,004	8712,514	12130,012	40312,434
3	38,063	45,488	205,590	690,107	1094,833	7626,154	10571,679	34952,482
4	50,417	57,286	207,002	663,794	1045,868	7123,894	9763,490	32602,002
5	66,343	73,325	221,385	661,113	1014,188	6601,870	9032,082	29817,517
6	85,684	91,954	237,197	658,852	1011,664	6456,725	8830,030	29156,784
7	115,824	120,965	254,055	666,979	1016,844	6259,459	8420,629	27842,375

Fonte: Autoria própria.

Tabela 24 – Custo energético (*Ws*) do Caixeiro Viajante no *cluster* Cubietruck

Nodos	eil76	kroA100	kroA200	a280	lin318	att532	rat575	rat783
0	2,880	8,040	77,765	266,965	474,175	3159,915	4312,140	14515,780
1	7,850	15,110	85,090	274,790	440,990	3093,860	4294,770	14289,130
2	13,410	18,345	73,455	219,105	340,110	2328,600	3191,985	10703,220
3	18,680	24,980	83,400	208,020	318,360	2087,740	2851,560	9519,720
4	23,625	29,450	101,300	210,275	310,575	1994,225	2665,825	8959,925
5	32,310	35,010	93,720	221,280	311,880	1922,790	2629,860	8847,120
6	35,245	41,195	97,545	221,480	311,325	1866,305	2540,510	8488,200
7	42,560	51,640	137,720	237,320	340,240	1841,280	2511,680	8275,240

Fonte: Autoria própria.

A Figura 22 apresenta os dados das Tabelas 22, 23 e 24 de forma gráfica para cada tipo de entrada para o problema do Caixeiro Viajante.

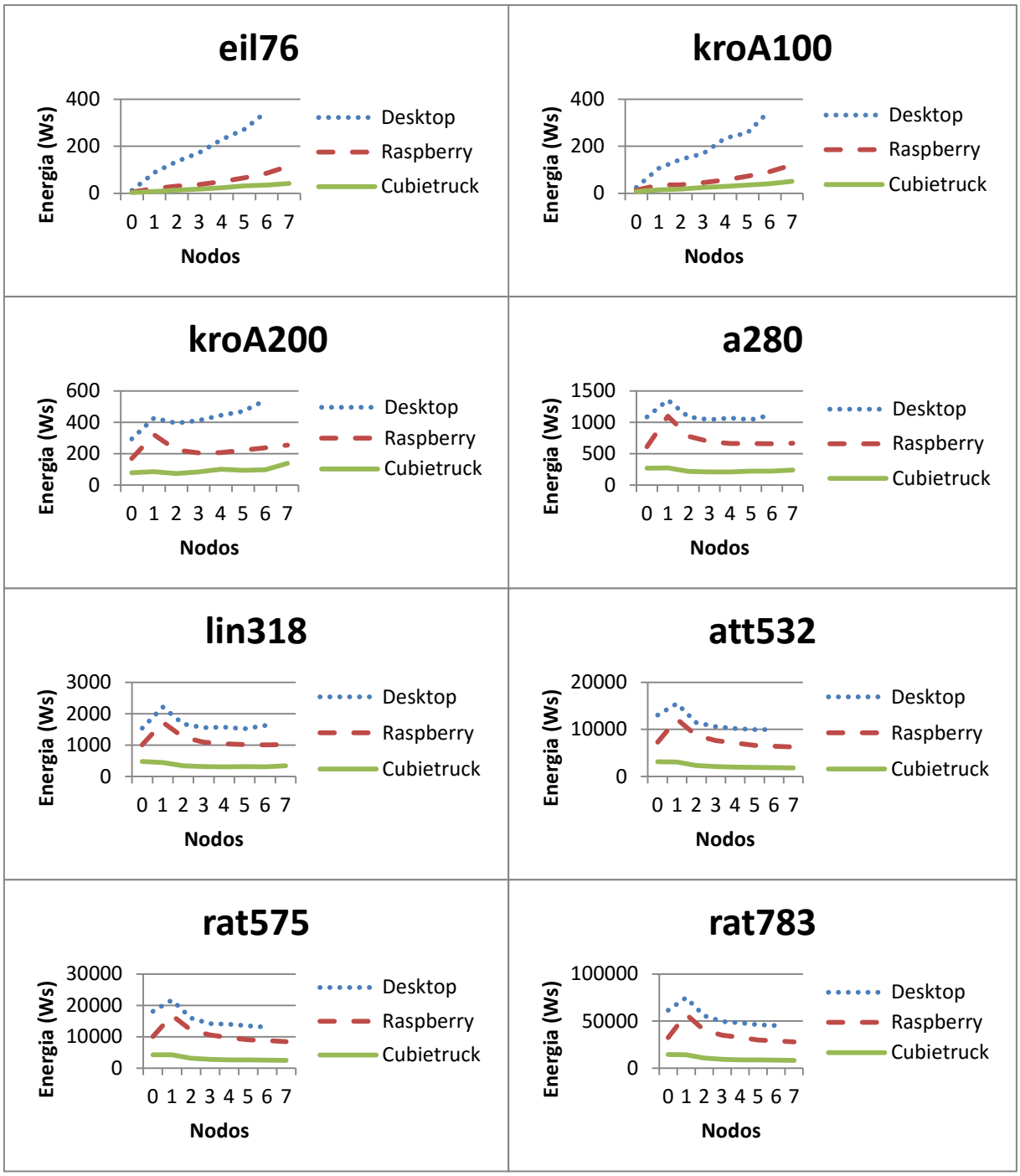

Figura 22 – Gráficos, resultados custo energético para Caixeiro Viajante
Fonte: Autoria própria.

Assim como no caso anterior, nota-se o pico de energia relacionado ao *cluster* Raspberry quando o *cluster* utiliza um nodo. Entretanto diferente do consumo energético para o problema de Multiplicação de Matriz, o *cluster* Raspberry possui melhor desempenho do que o *cluster desktop* para os problemas do Caixeiro Viajante. Para este problema o *cluster desktop* possui o pior desempenho energético, sendo o *cluster* Cubietruck com o melhor desempenho.

Observando os dados de *speedup*, verifica-se que o algoritmo para solucionar o problema do Caixeiro Viajante possui menor taxa de *speedup* do que o algoritmo para solucionar o problema de Multiplicação de Matriz e, como dito anteriormente, isso acaba impactando diretamente no consumo de energia. Dessa forma, como o ganho do paralelismo não incrementa o suficiente para suprir o consumo de energia gasto, o *cluster desktop* tenderá a não conseguir um melhor desempenho energético que os outros *clusters*.

6 CONCLUSÃO

Neste trabalho foram construídos três *clusters* (*desktop*, Raspberry e Cubietruck) com diferentes tipos de arquiteturas para coletar dados de custo energético, financeiro e velocidade de processamento.

Para medir a velocidade de processamento foram utilizadas duas métricas, *speedup* e o *benchmark High Performance Linpack* (HPL), sendo que, para calcular o *speedup,* foram necessários codificar algoritmos paralelos e sequenciais. Dois problemas foram selecionados para serem processados pelos *clusters*: Multiplicação de Matriz e Caixeiro Viajante, sendo que para realizar a paralelização e distribuição dos códigos entre as máquinas pertencentes ao *cluster* foi utilizada a biblioteca MPI de comunicação. Já para medir a energia gasta e armazenar os dados coletados, foi necessário construir um pequeno dispositivo utilizando um microcontrolador Tiva para conversão e envio dos dados a um computador. Para o computador foi desenvolvido um programa em Java para realizar comunicação com o microcontrolador e o *cluster*, com propósito de coletar energia somente enquanto o *cluster* estivesse processando.

Ao analisar os primeiros dados do HPL, tempo de processamento e *speedup*, fica evidente a discrepância de poder de processamento entre os dispositivos *desktops* e os dispositivos ARM, sendo que um único computador *desktop* consegue melhores resultados do que os outros *clusters* com todos os nodos. Entretanto, ao analisar o custo energético de cada *cluster*, pode-se observar que as Cubietrucks possuem o melhor desempenho em relação aos Raspberrys e Desktops, apesar desses dados serem estipulados e não devidamente mensurados. Pode-se observar também que, dependendo do problema, os Raspberrys utilizam menos energia do que os *desktops*.

Para finalizar, pode-se observar que dependendo do tipo de aplicação que se deseja processar é mais viável utilizar um *cluster* composto por dispositivos ARM, do que um *cluster* composto por computadores convencionais, pois o custo para montá-lo e mantê-lo é relativamente menor. Entretanto fica claro que o tempo e poder de processamento utilizado por cada um, são discrepantes, não sendo aconselhável o uso dos *clusters* Raspberry e Cubietruck para aplicações na quais o tempo seja característica crítica do sistema.

Apesar dos resultados obtidos neste trabalho mostrarem uma diferença grande no poder de processamento entre o *cluster desktop* e o *cluster* de dispositivos ARM, vale ressaltar que os dispositivos ARM estão evoluindo cada vez mais, com mais núcleos e mais memória. Como exemplo dessa evolução tem-se a placa Cubieboard4 (CUBIEBOARD4, 2015), a qual possui um processador ARM Cortex *octa-core*, além de um processador gráfico com 64 núcleos. Com esses novos dispositivos, novos estudos devem ser realizados, pois, com essa velocidade de evolução dos dispositivos ARM, é possível que o computador *desktop* se torne ultrapassado nas questões de poder de processamento e economia de energia.

REFERÊNCIAS

ACER. **Aspire MC605**. Disponível em: <http://www.acer.pt/ac/pt/PT/content/model-datasheet/DT.SM1EB.037>. Acesso em: 16 abr. 2014.

ADVANCED CLUSTERING TECHNOLOGIES. **How do I tune my HPL.dat file?**. Disponível em: <http://www.advancedclustering.com/act-kb/tune-hpl-dat-file/>. Acesso em: 28 abr. 2015.

AIEX, Renata. M.; BINATO, S.; RESENDE, Mauricio G. C. Parallel GRASP with Path-Relinking for job shop scheduling. **Parallel Computing**, v. 29, p. 393-430, 2002.

ALLEGRO MICROSYSTEMS. **Automotive Grade, Fully Integrated, Hall Effect-Based Linear Current Sensor IC with 2.1 kVRMS Voltage Isolation and a Low-Resistance Current Conductor**. Disponível em: <http://www.allegromicro.com/en/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS714.aspx>. Acesso em: 25 nov. 2014.

BADER, David A.; HART, William E.; PHILLIPS, Cynthia A. **Parallel Algorithm Design for Branch and Bound**. Tutorials on Emerging Methodologies and Applications in Operations Research, v. 76, pp. 5-5-44, 2005.

BALBINOT, Alexandre; BRUSAMARELLO, Valner João. **Instrumentação e fundamentos de medidas**. v. 2, Rio de Janeiro: LTC, 2007.

BLUM, Christian; ROLI, Andrea. Metaheuristics in Combinatorial Optimazation: Overview and Conceptual Comparison. **ACM Computing Surveys**, v. 35, p. 268-308, 2003.

COOK, Stephen A. An overview of computational complexity. **Communications of the ACM**, v. 26, n. 6, p. 401-407, 1983.

CORMEN, Thomas H. et al. **Algoritmos: Teoria e Prática.** 2ª ed., Rio de Janeiro: Elsevier, 2002.

COULOURIS, George F.; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas Distribuídos: Conceitos e Projeto.** 4ª ed., Porto Alegre: Bookman, 2007.

COX, Simon J. et al. Iridis-pi: a low cost, compact demonstration cluster. **Cluster Computing**, Southampton: Springer US, p. 1-10, 2013.

CUBIEBOARD. **CubieBoard: A series of open source hardware**. Disponível em: <http://cubieboard.org/>. Acesso em: 24 nov. 2014.

CUBIEBOARD4, **Cubieboard4 CC-A80 High-performance Mini PC Development Board Cubieboard A80**. Disponível em: < http://www.cubietruck.com/products/cubieboard4-cc-a80-high-performance-mini-pc-development-board>. Acesso em: 19 de jun. de 2015.

FEOFILOFF, Paulo. **Algoritmos Gulosos**. Disponível em: <http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html>. Acesso em: 20 mai. 2015.

FIALHO, Arivelto Bustamante. **Instrumentação Industrial: Conceitos, Aplicações e Análises**. 2ª ed., São Paulo: Érica, 2004.

FLYNN, Michael J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, v. 21, n. 9, p. 948-960, 1972.

GAREY, Michael R.; JOHNSON, David S. **Computers and Intractability: A guide to the Theory of NP-Completeness.** United States of America: Bell Telephone Laboratories, 1979.

GLOVER, Fred; KOCHENBERGER, Gary A. **Handbook of Metaheuristics**. International Series in Operations Research & Management Science, v. 57. Kluwer Academic Publishers, Norwell, MA, 2002.

INTEL. **Product and Performance Information**. Disponível em: <http://www.intel.com/content/www/us/en/gaming/overclocking-intel-processors.html>. Acesso em: 16 abr. 2014.

JÜNGER, Michael; REINELT, Gerhard; RINALDI, Giovanni. **The Traveling Salesman Problem**. Disponível em: <http://www.iasi.cnr.it/reports/R375/R375.html>. Acesso em: 1 ago. 2015. *Handbooks in Operations Research and Management Science*, 1994.

MORET, Bernard M. E. **The Theory of Computation.** 1ª ed., United States of America: Addison Wesley, 1997.

MUSSOI, Fernando Luiz R. **Sinais Senoidais: Tensão e Corrente Alternadas**. 3ª ed., Florianópolis: Centro Federal de Educação Tecnológica de Santa Catarina, 2006.

NOBRE, Ricardo Holanda. **Paralelismo como solução para redução de complexidade de problemas combinatoriais.** Mestrado Profissionalizante em Computação Aplicada - Instituto Federal de Educação, Fortaleza, 2011.

OU, Zhonghong et al. **Energy- and Cost-Efficiency Analysis of ARM-Based Clusters**. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2012.

PADOIN, Edson K.; OLIVEIRA, Daniel A. G. de; VELHO, Pedro; NAVAUX, Philippe O. A. **Evaluating Performance and Energy on ARM-based Clusters for High Performance Computing.** 41st International Conference on Parallel Processing Workshops, Pittsburgh, 2012.

PEGUERO, Braiam. **Setting Up an MPICH2 Cluster in Ubuntu**. Disponível em: < https://help.ubuntu.com/community/MpichCluster>. Acesso em: 28 abr. 2015.

PEREIRA, Fabio. **Microcontroladores MSP430: Teoria e Prática**. 1ª ed., São Paulo: Érica, 2005.

PETITET, A.; WHALEY, R. C.; DONGARRA, J.; CLEARY, A. **HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**. Innovative Computing Laboratory, University of Tennessee. Disponível em: <http://www.netlib.org/benchmark/hpl/>. Acesso em: 29 nov. 2014.

RASPBERRY PI. **Frequently Asked Questions**. Disponível em: <http://www.raspberrypi.org/help/faqs/>. Acesso em: 16 abr. 2014.

RAUBER, Thomas; RÜNGER, Gudula. **Parallel Programming for Multicore and Cluster Systems**. Germany: Springer, 2010.

REINELT, Gerhard. **TSPLIB**. Instituto de Ciência da Computação, Universidade de Heidelberg. Disponível em: <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>. Acesso em: 20 abr. 2015.

ROSE, César A. F.; NAVAUX, Philippe O. A.. **Arquiteturas Paralelas**. Porto Alegre: Bookman: Instituo de Informática da UFRGS, 2008.

SIPSER, Michael. **Introdução à Teoria da Computação.** 2ª ed., São Paulo: Thomson Learning, 2007.

SILVA JUNIOR, Vidal Pereira da. **Aplicações Práticas do Microcontrolador 8051**. 7ª ed., São Paulo: Érica, 1998.

SLOAN, Joseph D. **High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI.** United States of America: O'Reilly, 2004.

SINDI, Mohamad. **Top500 HPL Calculator**. Disponível em: <http://hpl-calculator.sourceforge.net/>. Acesso em: 28 abr. 2015.

TERADA, Routo. **Introdução à Complexidade de Algoritmos Paralelos.** São Paulo: IME-USP, 1990.

TEXAS INSTRUMENTS. **BeagleBone Black Development Board**. Disponível em: <http://www.ti.com/tool/beaglebk>. Acesso em: 16 abr. 2014.

TOCCI, Ronald J.; WIDMER, Neal S. **Sistemas Digitais: Princípios e Aplicações**. Rio de Janeiro: LTC, 2000.

TOSCANI, Laira V.; VELOSO, Paulo A. S. **Complexidade de Algoritmos: análise, projeto e métodos**. 3 ed., Porto Alegre: Bookman, 2012.

APÊNDICES

APÊNDICE A – Código do microcontrolador Tiva

```
//*****************************************************************************
//
// TCC_CurrentSensor.c - Project to read an analog value from an AC current
//sensor
//                       through the ADC and send via UART to PC.
//
//  This project aims to resolve the energy measurement problem to Trabalho
//de Conclusão de Curso
// of the Fabricio N. de Godoi the UTFPR, campus Pato Branco.
//
// Autor: Fabricio Negrisolo de Godoi
// Date: September 10, 2014
//
//*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/adc.h"
#include "driverlib/timer.h"
#include "utils/uartstdio.h"

//*****************************************************************************
//
//  ------ Ref. 0 A ------
//  [5V]
//  2,55 V --------- 3160 {ADC}
//
//  [3,3V]
//  1,68 V --------- 2079 {ADC}
//
//  RMS = sqrt( (Sum(ADC^2) / N )
//
//  -------------
//      Tiva    |
//              |
//         UART-|<---> PC
//              |
//           5V-|----> Sensor Vcc
//          GNC-|----> Sensor GND
//  ADC IN  PD0-|<---- Sensor with divider output (0 - 3.3V)
//              |
//  -------------
//
//      Sensor output w.o. div.  = 66     mV/A    (0-5   V)
//      Sensor output w.   div.  = 43.421 mV/A    (0-3.3 V)
//      ADC sensitive            = 0.8059 mV/ADC
//      Current sensitive = 18.559 mA/ADC | 1/54= 0.018518 = ~18.559 mA/ADC
//      Obs.: ADC - Natural value of ADC.
```

```
//
//  Iac = (ADC-2079)*19 {mA}
//  Icc = RMS(Iac)
//  Icc = sqrt(Sum(Iac^2)/N)
//  Icc= sqrt( Sum( (ADC-2079)*19 ) / N )
//*****************************************************************************

//*****************************************************************************
// Functions prototypes:
//*****************************************************************************
void UARTInit    (void);
void ADCInit     (void);
void TimerInit   (void);
void ADCSample   (void);

//*****************************************************************************
// Global variables:
//  samples: Storage array to ADC values.
//  sampleIndex: Index to samples array.
//  getADC: Variable to read ADC buffer.
//*****************************************************************************
uint32_t sample;
uint32_t samplesVector[4];
uint32_t samplesIndex = 0;
uint32_t samplesSum = 0;
int samplesCount = 0;
uint32_t getADC[1];

char send = 0;
char sendADC = 0;

//*****************************************************************************
// Start a new sample from ADC0 and send in UART0.
void
ADCSample(void){

    // Trigger the ADC conversion.
    ROM_ADCProcessorTrigger(ADC0_BASE, 3);
    // Wait for conversion to be completed.
    while(!ROM_ADCIntStatus(ADC0_BASE, 3, false));
    // Clear the ADC interrupt flag.
    ROM_ADCIntClear(ADC0_BASE, 3);
    // Read ADC Value.
    ROM_ADCSequenceDataGet(ADC0_BASE, 3, getADC);
    // Get sample.
    sample = getADC[0] - 2083; //2083 = 1,673*4095/3,3
    // Add the sample in the sum.
    samplesSum += (sample*sample);
    // Samples counter
    samplesCount++;
}

//*****************************************************************************
// This example demonstrates how to send a string of data to the UART.
//*****************************************************************************
int
main(void){

    // Enable lazy stacking for interrupt handlers.  This allows floating-
point
    // instructions to be used within interrupt handlers, but at the
```

```
expense of
    // extra stack usage.
    ROM_FPUEnable();
    ROM_FPULazyStackingEnable();

    // Set the clocking to run directly from the crystal.
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    // Enable the peripherals used by this project.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);    //UART
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);    //UART - PinOut
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);     //ADC
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);   //Timer0
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);   //Timer1

    // Enable processor interrupts.
    ROM_IntMasterEnable();

    //Peripherals initialization functions.
    UARTInit();     // UART configuration.
    ADCInit();      // ADC configuration.
    TimerInit();    //Timer configuration.

    // Loop forever.
    while(1){
    }
}
//*****************************************************************************
// RTIs functions:
//*****************************************************************************
// UART interrupt handler - echo.
void
UARTIntHandler(void)
{
    uint32_t ui32Status;
    char readUART;

    // Get the interrrupt status.
    ui32Status = ROM_UARTIntStatus(UART0_BASE, true);

    // Clear the asserted interrupts.
    ROM_UARTIntClear(UART0_BASE, ui32Status);

    // Loop while there are characters in the receive FIFO.
    while(ROM_UARTCharsAvail(UART0_BASE))
    {
    // Read the next character from the UART and write it back to the UART.
        readUART = ROM_UARTCharGetNonBlocking(UART0_BASE);

        if(readUART == 'p') //Stop sending ADC values.
            send = 0;
        else if(readUART == 's') //Start sending ADC values.
            send = 1;
        else if(readUART == 'a') //Alternate betewen ADC value and RMS
value.
            sendADC ^= 1;
    }
}
//Timer A0 interrupt handler - ~200 ADC samples
void TimerA0IntHandler (void){
```

```
    // Clear the timer interrupt flag.
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Start ADC conversion.
    ADCSample();

}
//Timer A1 interrupt handler - 60 Hz
void TimerA1IntHandler (void){

    // Clear the timer interrupt flag.
    ROM_TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

    // Send RMS via UART.
    // The receiver must do the following operation to get the RMS:
    // RMS = ( (Value Received)^(1/2) )*0.018559
    if(send == 1){
        if(sendADC == 0)UARTprintf("%d\n", (samplesSum/samplesCount));
        else UARTprintf("%d - %d\n",getADC[0], samplesCount);
        // Clear Counter.
        samplesCount = 0;
    }
    // Clear sum.
    samplesSum = 0;
}
//*****************************************************************************
// Initialization functions:
//*****************************************************************************
// UART0 initialization.
void
UARTInit(void){

    // Set GPIO A0 and A1 as UART pins.
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // Configure the  UART for 115,200, 8-N-1 operation.
    ROM_UARTConfigSetExpClk(UART0_BASE, ROM_SysCtlClockGet(), 115200,
                            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                             UART_CONFIG_PAR_NONE));

    // Configure UARTStdio to use UARTprintf
    // UART0, BAUD, Clock
    UARTStdioConfig(0,115200, 16000000);

    // Enable the UART interrupt.
    ROM_IntEnable(INT_UART0);
    ROM_UARTIntEnable(UART0_BASE, UART_INT_RX);
}
// ADC0 initialization.
void
ADCInit(void){

    // For this project ADC0 is used with AIN5 on port D2.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    // Select the analog ADC function for these pins.
    ROM_GPIOPinTypeADC(GPIO_PORTD_BASE, GPIO_PIN_2);
   //Enable sample sequence 3 with a processor signal trigger.  Sequence 3
   //will do a single sample when the processor sends a signal to start the
```

```
   //conversion.  Each ADC module has 4 programmable sequences, sequence 0
   //to sequence 3.
    ROM_ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
    //Oversample average (2, 4, 8, 16, 32, 64).
    ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);
  //Configure step 0 on sequence 3.  Sample channel 0 (ADC_CTL_CH5) in
  //single-ended mode (default) and configure the interrupt flag
  //(ADC_CTL_IE) to be set when the sample is done.  Tell the ADC logic
  //that this is the last conversion on sequence 3 (ADC_CTL_END).  Sequence
  //3 has only one programmable step.  Sequence 1 and 2 have 4 steps, and
  //sequence 0 has 8 programmable steps.  Since we are only doing a single
  //conversion using sequence 3 we will only configure step 0.  For more
  //information on the ADC sequences and steps, reference the datasheet.
   ROM_ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH5 | ADC_CTL_IE |
                                ADC_CTL_END);
    // Since sample sequence 3 is now configured, it must be enabled.
    ROM_ADCSequenceEnable(ADC0_BASE, 3);
    // Clear the interrupt status flag.  This is done to make sure the
    // interrupt flag is cleared before we sample.
    ROM_ADCIntClear(ADC0_BASE, 3);
}
//Timer initialization.
void TimerInit (void){

////////////////////////////////////////////////////////////////////////////
    // TimerA0:
    // Configure Timer0A as a 32-bit periodic timer.
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    // Set the Timer0A load value to 12000 Hz = 200*60Hz.
    // Voltage frequency = 60 Hz
    // ADC samples per cycle (60Hz) = ~200
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ROM_SysCtlClockGet()/12000 );
    // Enable the Timer0A interrupt on the processor (NVIC).
    ROM_IntEnable(INT_TIMER0A);
    // Configure the Timer0A interrupt for timer timeout.
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    // Enable Timer0A.
    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
////////////////////////////////////////////////////////////////////////////
    // TimerB1: //Frequencymeter
    // Configure Timer1A as a 32-bit counter timer.
    ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
    // Set the Timer1A send RMS value (60 Hz).
    ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, ROM_SysCtlClockGet()/ 60 );
    // Enable the Timer1A interrupt on the processor (NVIC).
    ROM_IntEnable(INT_TIMER1A);
    // Configure the Timer1A interrupt for timer timeout.
    ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
    // Enable Timer1A.
    ROM_TimerEnable(TIMER1_BASE, TIMER_A);
}
```

ANEXOS

ANEXO A – Exemplo de entrada para Multiplicação de Matriz

```
8
8
5.935972;0.148261;5.968474;9.638044;7.190215;9.675515;7.980445;9.043702;
4.587532;7.810068;9.495145;4.255758;0.576350;4.374937;5.309189;8.219427;
8.514342;1.542812;8.317196;3.100153;3.569919;0.351970;1.173154;6.909545;
1.199568;1.437995;9.635937;7.168042;1.076038;6.826152;6.843557;9.056483;
2.925637;0.457387;4.978712;2.420782;4.713145;5.555062;6.795718;0.022335;
0.011968;2.288832;2.283424;8.329164;5.388985;5.853343;8.681134;6.562139;
6.049816;3.962455;5.621151;5.685753;1.130497;6.697189;2.511906;7.974054;
5.142698;8.679310;8.839147;0.121410;1.100092;3.552292;5.676472;7.895810;
```

ANEXO B – Exemplo de entrada para o problema do Caixeiro Viajante

```
26
1 37 52
2 49 49
3 52 64
4 20 26
5 40 30
6 21 47
7 17 63
8 31 62
9 52 33
10 51 21
11 42 41
12 31 32
13 5 25
14 12 42
15 36 16
16 52 41
17 27 23
18 17 33
19 13 13
20 57 58
21 62 42
22 42 57
23 16 57
24 8 52
25 7 38
26 27 68
```