

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

HELMAR ALVARES

*AMBIENTE WEB PARA SIMULAÇÃO DE ALGORITMOS PARALELOS
PARA PROBLEMA DE ESCALONAMENTO “JOB SHOP”*

Rio de Janeiro
2010

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

HELMAR ALVARES

*AMBIENTE WEB PARA SIMULAÇÃO DE ALGORITMOS PARALELOS
PARA PROBLEMA DE ESCALONAMENTO “JOB SHOP”*

Dissertação de Mestrado Submetida ao Programa de Pós-Graduação em Informática, Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro. como parte dos requisitos necessários para obtenção do Título de Mestre em Informática.

Orientadora: Eliana Prado Lopes Aude, Ph.D. (UFRJ)

Co-orientador: Ernesto Prado Lopes, Ph.D. (UFRJ)

Rio de Janeiro
2010

FICHA CATALOGRÁFICA

A473 Alvares, Helmar.

Ambiente *web* para simulação de algoritmos paralelos para problema de escalonamento job shop. / Helmar Álvares – 2010.
129 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Núcleo de Computação Eletrônica, 2010.

Orientadora: Eliana Prado Lopes Aude
Co-orientador Ernesto Prado Lopes

1. Simulador – Teses. 2. Algoritmo – Teses. 3. Arquitetura Paralela – Teses. 4. Escalonamento – Teses. 5. Job Shop – Teses. I. Eliana Prado Lopes Aude (Orient.). II. Ernesto Prado Lopes (Co-orient.). III. Universidade Federal do Rio de Janeiro. Instituto de Matemática. Núcleo de Computação Eletrônica. IV. Título.

CDD

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS MATEMÁTICAS E DA NATUREZA
INSTITUTO DE MATEMÁTICA

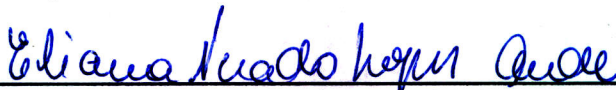
**AMBIENTE WEB PARA SIMULAÇÃO DE ALGORITMOS PARALELOS PARA
PROBLEMA DE ESCALONAMENTO “JOB SHOP”.**

POR

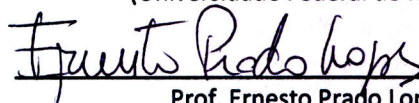
HELMAR ALVARES

Dissertação de Mestrado Submetida ao Programa de Pós-graduação em Informática do Instituto de Matemática / Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro – UFRJ, como parte dos requisitos necessários para obtenção do Título de Mestre em Ciências em Informática.

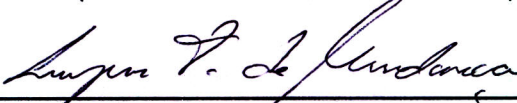
Aprovada em:



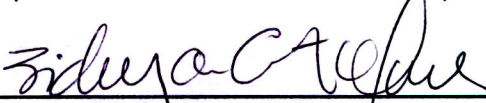
Profª. Eliana Prado Lopes Aude, Ph.D.
(Universidade Federal do Rio de Janeiro)



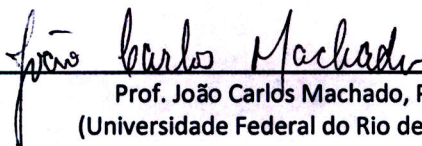
Prof. Ernesto Prado Lopes, Ph.D.
(Universidade Federal do Rio de Janeiro)



Profª. Luziane Ferreira de Mendonça D.Sc.
(Universidade Federal do Rio de Janeiro)



Prof. Sidney de Castro Oliveira D.Sc.
(Universidade Federal do Rio de Janeiro)



Prof. João Carlos Machado, Ph.D.
(Universidade Federal do Rio de Janeiro)

RIO DE JANEIRO
20 de ABRIL 2010

“verum ipsum factum”

- o conhecimento vem pela prática -

Giambattista Vico – filósofo Italiano

(1668 - 1744)

AGRADECIMENTOS

Nestes últimos três anos, muitas pessoas de forma ativa ou indireta participaram e colaboraram para o sucesso desta dissertação e para o meu aprimoramento pessoal.

Gostaria de expressar minha gratidão aos professores Eliana Prado Lopes Aude e Ernesto Prado Lopes, que com muita paciência e inteligência me orientaram na presente dissertação de mestrado, dedicando muitas horas de trabalho as discussões, validações e contribuições para o crescimento das idéias do objeto desta dissertação. Não poderia deixar de agradecer aos professores Gabriel P. Silva e Lilian Markenzon pelos valiosos ensinamentos, os quais contribuíram para a fundamentação das pesquisas.

Agradeço aos meus colegas do laboratório de robótica CONTROLAB da UFRJ, onde sempre fui incentivado nas pesquisas, e que forneceram valiosas informações aos estudos necessários ao desenvolvimento do Simulador. Gostaria de citar especialmente o colega Cristiano Aguiar pelas orientações e incentivos à continuidade do curso de mestrado, hora findo, como mais uma etapa de minha vida.

Minha eterna gratidão à minha família pelo apoio e por terem ficado com o aspecto mais tortuoso, suportando a minha ausência e me incentivando nos momentos em que eu estava presente.

Por fim, gostaria de agradecer, de forma indistinta, a todos os membros da banca examinadora pela leitura crítica e cuidadosa deste trabalho e pelos comentários e correções que me foram encaminhados, os quais foram fundamentais para a melhoria desta pesquisa.

RESUMO

ALVARES, Helmar. **Ambiente *web* para simulação de algoritmos paralelos para problema de escalonamento job shop**. 2010. 129 p. Dissertação (Mestrado em Informática) – Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2010.

Neste trabalho é proposto um ambiente web para simulação de algoritmos paralelos dedicados à solução do problema de escalonamento de tarefas conhecido como “Job Shop Problem” (JSP). O ambiente disponibiliza uma plataforma com diferentes linguagens de processamento e faz uso de uma placa gráfica (“Graphics Processing Unit” (GPU)) de baixo custo e alto desempenho conectada a um servidor de aplicação JAVA para auxiliar o projeto de soluções mais eficientes para o JSP. A aplicabilidade da teoria não se restringe apenas ao setor industrial, como por exemplo a fabricação de chips e linhas de produção, mas também a diversas outras áreas essenciais como a robótica, transporte e comunicações. Dentro de um único ambiente web, diferentes métodos e metas-heurísticas podem ser analisados e comparados. Cada novo método desenvolvido é armazenado no ambiente e disponibilizado para todos os seus usuários. Este acúmulo de métodos de solução, em acesso aberto, permite convergir para soluções ótimas para o JSP. A meta-heurística Busca Tabu foi desenvolvida e disponibilizada no sistema como estudo de caso. O algoritmo usa a arquitetura paralela do sistema para manipular threads. Cada thread representa uma operação de um Job e o número de threads escala com o tamanho do problema. As threads trocam informações na procura da melhor solução. Esta cooperação decresce o tempo de resposta de uma ou duas ordens de grandeza.

Palavras chaves:

- *Escalonamento Job Shop; Simulador web; JSP; CUDA; GPU; SIMD; thread; Grafo Disjuntivo; Busca Tabu.*

ABSTRACT

ALVARES, Helmar. **Ambiente *web* para simulação de algoritmos paralelos para problema de escalonamento job shop**. 2010. 129 p. Dissertação (Mestrado em Informática) – Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2010.

This work proposes a WEB-Based simulation environment for parallel algorithms which solves scheduling problems known as Job Shop Problems (JSPs). The environment supports multi-language platforms and uses a low cost, high performance Graphics Processing Unit (GPU) connected to a Java application server to help design more efficient solutions for JSPs. The applicability of the theory is not restricted to the industrial sector, such as chip manufacturing and production lines, but includes several other keys areas such as robotics, transport and communications. Within a single web environment one can analyze and compare different methods and metaheuristics. Each newly developed method is made stored in an environment library and made available to all other users of the environment. This amassment of openly accessible solution methods will allow for the rapid convergence towards optimal solutions for JSPs. A Tabu Search meta-heuristics was developed and made available to system users as a case study. The algorithm uses the parallel architecture of the system to handle threads. Each thread represents a job operation and the number of threads scales with the problem's size. The threads exchange information in order to find the best solution. This cooperation decreases response times by one or two orders of magnitude.

Keywords:

- *Job Shop Scheduling Problem; Web Simulator; JSP; CUDA; GPU; SIMD; thread; Disjunctive Graph; Tabu Search.*

Lista de Figuras

Figura 1 - Representação de um problema JSP através de Grafo Disjuntivo	32
Figura 2 - Uma solução viável para o problema JSP definido na Tabela 1	33
Figura 3 - Representação da seqüência de escalonamento das operações de cada tarefa, para as máquinas definidas na Tabela 1, demonstrando uma das possíveis soluções viáveis ao problema JSP.....	34
Figura 4 - Gráfico de Gantt gerado pelo simulador da presente dissertação, descrevendo outra solução viável ao problema JSP definido na Tabela 1.....	35
Figura 5 - Gráfico de Evolução gerado pelo Simulador para o problema MT10	36
Figura 6 - Identificação do tempo inicial e final, por uma <i>thread</i> que representa um nó do grafo G.....	42
Figura 7 - Exemplo de busca pela vizinhança (<i>Neighborhood</i>) pela reversão entre arcos de duas operações subsequentes, O_{11} (a) e O_{21} (b).....	44
Figura 8 - Caminho crítico encontrado em uma solução viável do problema, descrito na Tabela 1, na cor cinza	45
Figura 9 - Exemplo de perturbação no caminho do grafo G através da troca da operação O_{11} pela sua operação precedente O_{21} pertencentes ao caminho crítico.....	50
Figura 10 - Exemplo de resultado entre o movimento entre as operações O_{11} e O_{21} pertencentes ao caminho crítico	50
Figura 11 - Modelo de arquitetura SISD	59
Figura 12 - Modelo de arquitetura SIMD.....	60
Figura 13 - Modelo de arquitetura MISD.....	61
Figura 14 - Modelo de arquitetura MIMD	62
Figura 15 - Diferenças entre a distribuição dos componentes em uma CPU e em uma GPU (NVIDIA, 2008).....	66
Figura 16 - Arquitetura da família GeForce G80 (KANTER, 2008).	68
Figura 17 - Arquitetura de uma <i>Streaming</i> Processor (SP). (KANTER, 2008).....	69
Figura 18 - Hierarquia de memória de uma Geforce G80 (NVIDIA, 2008)	70
Figura 19 - Abstração para distribuição das <i>threads</i> (NVIDIA, 2008)	75
Figura 20 - Pilha de <i>software</i> da arquitetura do SDK (NVIDIA, 2008)	78
Figura 21 - Fluxo do processo de compilação – <i>nvcc</i> (MURPHY, 2008)	80
Figura 22 - Arquitetura do Simulador <i>web</i> para problemas JSP	90
Figura 23 - Arquitetura do Servidor de Métodos	92
Figura 24 - Configuração do compilador <i>nvcc</i> na IDE	105
Figura 25 - Configuração dos diretórios de inclusão e as <i>macros</i> utilizadas no projeto.....	105

Figura 26 - Três projetos do simulador criados no ambiente da IDE e detalhe ao lado do projeto “tsjbs”	107
Figura 27 - Arquitetura em camadas de software do simulador.....	109
Figura 28 - Módulos do <i>software</i> que compõem a <i>interface C</i>	110
Figura 29 - Módulos do <i>software</i> que controla o dispositivo GPU	111
Figura 30 - Página inicial do Simulador <i>web</i>	112
Figura 31 - Detalhes das caixas de diálogo para dados de entrada, número de <i>job</i> e número de Robôs	113
Figura 32 - Detalhes das caixas de diálogo para dados de entrada, descrição da primeira tarefa.....	113
Figura 33 – Detalhes das caixas de diálogo para dados de entrada, descrição dos pesos	113
Figura 34 - Detalhes das caixas de diálogo para dados de entrada	114
Figura 35 - Página de conferência dos dados do problema e envio	115
Figura 36 - Link na página inicial para upload de problemas.....	115
Figura 37 - Página de <i>upload</i> de problemas.....	116
Figura 38 - Página inicial com problemas clássicos, exemplo	116
Figura 39 - Documento PDF descrevendo uma solução viável	117
Figura 40 - Exemplo para dados do problema e medidas da solução	118
Figura 41 - Dados de entrada do problema	118
Figura 42 - Link ao arquivo XML armazenado no servidor de aplicação	118
Figura 43 - Gráfico de linha demonstrando a evolução por iterações do algoritmo “6 by 6 test case from Muth and Thompson”	120
Figura 44 - Diagrama de Gantt descrevendo uma solução viável	120
Figura 45 – Tempo computacional por <i>threads</i>	122

Lista de Tabelas

Tabela 1- Exemplo para a ordem de operações e duração de operações para cada tarefa.....	33
Tabela 2- Estrutura de dados de entrada do problema JSP com índice da tabela id da <i>thread</i>	47
Tabela 3- Estrutura de dados de saída ou solução, à esquerda. Neste exemplo é demonstrada uma solução viável para o problema JSP da Tabela 2.....	48
Tabela 4- Estrutura de dados da lista tabu. Neste exemplo é demonstrado o estado de uma solução viável para o problema JSP descrito na Tabela 2.....	49
Tabela 5- Principais modelos da família da NVIDIA.....	67
Tabela 6- Tipos de memória da arquitetura de memória GeForce G80	70
Tabela 7- Tipos de dados primitivos equivalentes entre C e Java.....	100
Tabela 8- Resultados para 08 instâncias com diferentes tamanhos de entrada.....	123

Lista de Abreviaturas e Siglas

ANSI.....	American National Standards Institute
API	Application Programming Interface
CSS.....	Cascading Style Sheets
CUBLAS.....	Compute Unified Basic Linear Algebra
CUDA.....	Compute Unified Device Architecture
CUFFT	Compute Unified Fast Fourier Transform
DOM.....	Document Object Model
EDG	Edson Design Group
FLOP.....	FLoating point Operations Per second
GNU	General Public License
HLSL	High Level Shader Language
HTML	HyperText Markup Language
HTTP.....	The Hypertext Transfer Protocol
IDE	Integrated Development Environment
IEEE.....	Institute of Electrical and Electronics Engineers
IP.....	Internet Protocol
J2EE	Java 2 Enterprise Edition
JIT	Just in time
JNA	Java Native Access
JSE	Java Standard Edition
JSP	Java Server Pages
LB.....	Lower Bound
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MRE	Mean Relative Error
OCG.....	Optimized Code Generator
PCIe	Peripheral Component Interconnect Express
PTX.....	Parallel Thread eXecution
SAX.....	Simple API for XML
SDK.....	Software Development Kit
SFU.....	Special Function Unit
SIMD	Single Instruction Multiple Data
SISD.....	Single Instruction Single Data
SLI.....	Scalable Link Interface
SM.....	Streaming Multiprocessor
SP	Streaming Processor
SPARC.....	Scalable Processor ARChitecture
SRAM.....	Static Random Access Memory
T&L	Transform and Lighting
TPC.....	Thread Processing Cluster
UB	Upper Bound
URL.....	Uniform Resource Locator
XML.....	eXtensible Markup Language

Sumário

Capítulo 1

Introdução	16
1.1 Apresentação.....	16
1.2 Caracterização do Problema	17
1.3 Objetivo	18
1.4 Contribuições.....	20
1.5 Organização	22

Capítulo 2

Fundamentação Teórica	24
2.1 Definição do Problema “Job Shop”	24
2.2 Estado da Arte	25
2.3 Métodos de resolução	28
2.3.1 Métodos Exatos	28
2.3.2. Métodos aproximativos	29
2.3.2.1. Heurísticas.....	29
2.3.2.2. Metas-Heurísticas.....	29
2.4. Instâncias de problemas JSP	31
2.5. Grafo Disjuntivo	31
2.6. Gráfico de Gantt	34
2.7. Gráfico XY - Evolução	35

Capítulo 3

Algoritmo - Busca Tabu Paralelo	37
3.1 Introdução.....	37
3.2 Motivação	38
3.3 Busca Tabu (TS).....	39
3.4 Busca Tabu Paralela	39
3.4.1 Heurística Paralela.....	40
3.4.2 Caminho Critico	45
3.4.3 Estrutura de Vizinhaça para o Problema JSP	46
3.4.4 Lista de Entrada.....	46
3.4.5 Lista de Saída.....	47
3.4.6 Lista tabu	48
3.4.7 Critério de Aspiração	50
3.4.8 Movimentação entre operações.....	51
3.4.9 Solução Inicial.....	51
3.4.10 Solução vizinha.....	52
3.4.11 Solução candidata	53
3.4.12 Critério de Parada.....	54
3.4.13 Invocação do Kernel pelo CUDA	55

Capítulo 4

Computação Paralela - CUDA.....	57
4.1.Introdução.....	57
4.2.Motivação	58
4.3 Taxonomia de Flynn	59
4.4 Gerações Anteriores de Programação para GPU.....	62

4.5 GPU "many core"	64
4.6 Arquitetura em threads	66
4.7 Arquitetura da GPU G80.....	68
4.8 Hierarquia de Memória	70
4.8.1 Registrador local por thread.....	71
4.8.2 Memória compartilhada por bloco.....	72
4.8.3 Memória Constante e Textura por grid	72
4.8.4 Memória Global por grid	72
4.8.5 Coalescência de memória	73
4.9 Grids de Threads	74
4.10 API de Alto e Baixo Nível em uma GPU	77
4.11 CUDA	77
4.11.1 Arquitetura de Pilha de Software	77
4.11.2 C/C++ Linguagem para GPU.....	79
4.11.3 Compilador	79
4.11.3.1 Opções do compilador nvcc.....	81
4.11.4 Extensões da Linguagem	81
4.11.5 Variáveis Globais.....	85
4.11.6 Função de Sincronização	85
4.11.7 Função Atômica.....	85
4.11.8 Gerenciamento de Memória	86

Capítulo 5

Simulador web - arquitetura	87
5.1. Introdução.....	87
5.2. Motivação	87
5.3. Arquitetura do simulador.....	89
5.4. Arquitetura em Camadas.....	91
5.4.1. Servidor web.....	93
5.4.2. Java Server Pages	94
5.4.3. Classe Servlet.....	95
5.4.4. Classes JavaBeans	97
5.5. Mensagens	97
5.5.1. API SAX.....	98
5.5.2. Acessando Bibliotecas Nativas.....	99
5.5.3. Tipos de Dados compartilhados	100
5.5.4. Objeto compartilhado.....	101
5.6. Ambiente integrado de desenvolvimento	102
5.6.1 Instalação do Netbeans	104
5.6.2 Abrindo um novo projeto	106
5.6.3 Projetos para o simulador	106
5.6.4 Ambiente Web e Java.....	107
5.6.5 Ambiente C - Interface JNA	109
5.6.6 Ambiente CUDA	110
5.6.7 Escrever um novo Método utilizando Templates	111
5.7. Funcionalidades.....	112
5.7.1. Escrever uma nova Instância	113
5.7.2. Enviar uma Instância - upload	115
5.7.3. Instâncias Clássicas	116
5.7.4. Solução em PDF.....	116
5.8. Estrutura do Arquivo de Entrada para Upload	121

Capítulo 6

Experimentos e Resultados.....122

6.1. Experimentos.....122

Capítulo 7

Conclusão124

7.1. Conclusão.....124

7.2. Limitações e Trabalhos Futuros.....126

Referências Bibliográficas.....127

CAPÍTULO 1

Introdução

1.1 Apresentação

A indústria está sempre procurando diferentes meios para aperfeiçoar a sua produtividade através do planejamento e controle da produção na utilização das máquinas e tempo-ciclo de produto. Em vista disso, a gestão tecnológica é essencial para manter a indústria competitiva em um mundo economicamente globalizado. Desta forma, um escalonamento de tarefas ótimo no uso destas máquinas é fundamental para proporcionar maior evolução na produtividade desta indústria.

A atividade de escalonamento em um ambiente de produção em grande escala pode ser definida como a atividade de distribuição de tarefas, ou tarefas para recursos durante um tempo, para alcançar determinados objetivos. Neste contexto, o escalonamento é a distribuição de tarefas compartilhadas no tempo para atividades competitivas, e na terminologia da indústria, as tarefas (*jobs*) são as atividades e as máquinas ou robôs representam os recursos.

Um dos modelos mais conhecidos da teoria de escalonamento é o *Job-Shop*, o qual é considerado uma boa representação do domínio geral, e tem conseguido reputação das mais difíceis de resolver entre conjunto de problemas de escalonamento NP-difícil.

A área de aplicabilidade da teoria não se restringe apenas ao setor industrial, como por exemplo, a fabricação de *chips* e a linha de produção, mas também a diversos outros serviços essenciais como o transporte, comunicações e robótica.

O escalonamento tem sido objeto de diversos estudos nos últimos 50 anos pela sua importância e ampla aplicabilidade industrial. O problema não é somente um problema de otimização combinatória, considerado como NP-difícil, mas é reconhecida a sua reputação de ser um dos problemas mais difíceis na sua resolução (GAREY et al., 1976).

1.2 Caracterização do Problema

Para auxiliar a tomada de decisão ou desenvolver controles para o escalonamento de tarefas na produção de uma indústria, seria necessário um sistema para a programação desta produção ou uma ferramenta de apoio de forma a reduzir seu custo operacional, aumentar o desempenho de produção e obter competitividade de mercado.

Este problema é conhecido como “*Job Shop Problem*” (JSP) onde o número de soluções possíveis para o escalonamento de tarefas apresenta uma explosão fatorial. Ou seja, este número pode chegar a $(n!)^m$ onde n é o número de tarefas (*jobs*) e m é o número robôs (máquinas).

Como não poderia deixar de ser, o paralelismo tem potencialmente importante consequência para a área de otimização combinatória. A utilização de processadores com arquitetura paralela torna possível determinar soluções em menor tempo. Entretanto, apresenta o problema da complexidade no desenvolvimeto do algoritmo para o escalonamento (“*scheduling*”) de tarefas, que envolve alocação de recursos no tempo com a finalidade de executar uma série de tarefas. Este problema é um grande desafio computacional, e os métodos anteriores não prevêem um algoritmo eficiente que garanta a obtenção de uma solução ótima em tempo polinomial para grandes problemas JSP.

Foram identificados no estado da arte diversos métodos, como método exato “*Branch and Bound*” (BB), metas-heurísticas aproximativas como recozimento simulado ou “*Simulated Annealing*” (SA), algoritmo genético ou “*Genetic Algorithm*” (GA) e busca-tabu ou “*Tabu Search*” (TS), entre outros.

Até o momento, o método que apresenta garantias de otimalidade são os exatos, por exemplo, “*Branch and Bound*” (CARLIER e PINSON, 1989). Mas esta solução apresenta alto custo computacional o que a inviabiliza para grandes instâncias de problemas.

Os métodos aproximativos melhoraram consideravelmente este custo computacional viabilizando o processamento de grandes instâncias de problemas, mas não garantem a otimalidade da solução. Porém foi demonstrado que estes métodos atingem a sub-otimalidade da solução o que satisfaz as necessidades da indústria.

O procedimento dos métodos ditos aproximativos consiste em utilizar uma heurística inteligente ao pesquisar o espaço de busca para encontrar a melhor solução dentre o conjunto de soluções do problema. A qualidade da solução é proporcional ao número de análises feitas pelo algoritmo e este processo é exaustivo diante das combinações possíveis e viáveis. Estas combinações crescem exponencialmente diante do tamanho da instância do referido problema e consomem tempo proporcional para encontrar uma solução ótima.

Ao se procurar a documentação destes métodos aproximativos na literatura, nem sempre estas estão suficientemente claras para uma aplicação imediata, atendem apenas a determinados tipos de problemas e requerem diversos ajustes de parâmetros de funções de acordo com este problema. É difícil fazer uma comparação entre estes métodos para uma avaliação, tendo em vista o objetivo de se atingir a otimalidade ou qualidade da solução e o tempo de resposta computacional necessário.

No desenvolvimento do planejamento e programação de algoritmos para soluções JSP, é razoavelmente comum assumir que os dados do problema são pré-definidos, fazendo uso dos diversos “*benchmarks*”, elaborados ao longo dos últimos trinta anos, por diversos pesquisadores deste problema. No entanto, os problemas estudados no processo de desenvolvimento destes algoritmos, são exatamente determinados o que não ocorre em situações do mundo real: as tarefas muitas vezes são maiores ou menores do que o esperado. Resolver um problema JSP otimamente é difícil, mas quando os dados de entrada pertencem a situações reais, estes são incertos e o problema fica muito mais complexo por conta da imprecisão da função objetivo, do tamanho do espaço de busca e da existência de múltiplos mínimos locais.

1.3 Objetivo

O ambiente *web* de desenvolvimento é importante para a eficiência no planejamento da programação da produção industrial. Nele, pesquisadores ou engenheiros podem melhor planejar a linha de produção compartilhando recursos e idéias.

Nesta dissertação é apresentado um Simulador em ambiente *web* dedicado ao estudo de algoritmos paralelos, apresentando uma infraestrutura paralela de baixo

custo, para fomentar soluções híbridas de processamento que sejam mais eficientes para a resolução dos problemas de otimização combinatória, mais especificamente ao problema de escalonamento “*Job Shop Problem*” (JSP). Este problema é conhecido como um dos mais difíceis problemas de otimização combinatória por possuir, nas aplicações práticas, números de restrições muito grandes.

Como o problema JSP é reconhecido como um dos problemas mais difíceis, um dos fatores importantes a considerar é o tempo de resposta do Simulador para resolução de grandes problemas JSP. Sendo assim, foi abordada uma solução híbrida de processamento serial e processamento paralelo com uma arquitetura de alto desempenho para atingir tempos de respostas aceitáveis.

Com este desafio, foi idealizado na arquitetura do servidor de aplicação, proposto nesta apresentação, um processamento paralelo de baixo custo. Este ambiente foi planejado para que se possa compartilhar pela *web* as diversas metas-heurísticas que venham a ser desenvolvidos para a resolução do problema JSP. Este recurso ajuda os engenheiros e pesquisadores na compreensão da dinâmica de cada uma das metas-heurísticas, seu compartilhamento, sua análise e aplicabilidade nos diversos ambientes de produção.

Sendo assim, um Simulador *web* foi criado como ferramenta de ensino e pesquisa com o objetivo de possibilitar aos pesquisadores compartilharem recursos computacionais no desenvolvimento e simulação de problemas JSP, mesmo para grandes instâncias, com entradas do problema para até cem máquinas (*robôs*) e cem tarefas (*jobs*).

Com a disponibilização de um mesmo ambiente na *web* para análise e comparação entre as diversas heurísticas ou metas-heurísticas, processados em uma arquitetura paralela, pode-se objetivar as seguintes características:

- Uma análise mais apurada para a resolução do problema JSP;
- Potencializar com o processamento paralelo o procedimento de identificação no espaço de busca da melhor solução;
- Reduzir o tempo de análise do algoritmo de busca.

A mudança de paradigma com a introdução da computação paralela acarreta na forma de resolver o problema onde os desenvolvedores, atualmente, estão

acostumados a pensar um algoritmo na sequência de passos, de forma *serial*, passo após outro. Para resolver este mesmo problema utilizando a referida arquitetura paralela, os desenvolvedores precisariam pensar o seu algoritmo em subdivisões do problema de forma simultânea. Na análise deste problema, é preciso determinar a melhor divisão dos dados a serem processados, ou nichos de dados para serem distribuídos nos diversos processadores paralelos.

Deste modo, outro objetivo almejado é disponibilizar um ambiente aos pesquisadores em um laboratório para o desenvolvimento de heurísticas ou metas-heurísticas híbridas (seriais e paralelos) para elaborar novas abordagens ao problema JSP.

Um grande desafio é para que este ambiente *web* forneça, com escalabilidade e alta disponibilidade, páginas geradas dinamicamente para um navegador *web* de busca o qual esteja acessando o referido sistema através de uma rede intranet.

Como não poderia deixar de ser, o paralelismo tem potencialmente importante consequência para a área de otimização combinatória. A utilização de processadores com arquitetura paralela torna possível determinar soluções em menor tempo. Entretanto, apresenta o problema da complexidade no desenvolvimento do algoritmo para o escalonamento.

1.4 Contribuições

A contribuição desejada é ajudar a desenvolver e testar novos métodos neste ambiente híbrido de processamento, com heurísticas ou metas-heurísticas e algoritmos paralelos, promovendo assim resoluções para o difícil problema JSP.

Os resultados deste esforço poderão ficar disponibilizados via *web* através de uma biblioteca de métodos para as comunidades interessadas neste problema, como os laboratórios das universidades e indústrias por exemplo.

Neste sentido, um ambiente de estudos com ferramentas apropriadas poderá colaborar no desenvolvimento de novos métodos seriais e paralelos para solucionar o problema JSP.

Motivado para contribuir na resolução do problema combinatorial JSP foi instalado e configurado um servidor de aplicação com arquitetura paralela de alto desempenho no laboratório de Automação e Robótica (CONTROLAB) do Núcleo de Computação Eletrônica da UFRJ. A tecnologia CUDA que será visto no Capítulo 4 é uma alternativa, a baixo custo, mais viável à utilização de supercomputadores. Neste início, o ambiente instalado possui 112 núcleos de processamento, com desempenho teórico máximo de 450 GFLOPs, mas pode ser facilmente expandido para 400 núcleos (processadores) ou mais, através de simples troca da GPU (placa gráfica) ou “clusterização” do servidor de aplicação.

A solução adotada e descrita na presente dissertação é a disponibilização de um *ambiente web para simulação*, instalado em um laboratório para o desenvolvimento de novas soluções de maneira a auxiliar a Indústria na tomada de decisão.

Neste ambiente, é fornecida ao pesquisador a facilidade de testar diferentes algoritmos de escalonamento para resolver o problema de alocar as operações necessárias para a execução de um conjunto de tarefas (*jobs*) aos robôs (máquinas) disponíveis, de forma a perseguir o tempo máximo para que todas as operações escalonadas sejam executadas e este tempo seja minimizado.

Devido à explosão de possíveis soluções ao problema JSP, os diferentes algoritmos de escalonamento, testados neste ambiente, são executados em uma placa de alto desempenho desenvolvida pela NVIDIA (Geforce 9800 GT). Esta placa funciona como um coprocessador paralelo e é utilizado no servidor de aplicação *web*. Este coprocessador é programado em linguagem C estendida que permite programação paralela com baixa curva de aprendizagem.

Neste contexto, grandes instâncias do problema JSP podem ser resolvidas satisfazendo os requisitos de rapidez relativos a cada aplicação e fazendo uso de tecnologia *web*.

Esta solução diminui o custo da manutenção, e torna possível a disponibilidade do sistema a vários pesquisadores de forma a viabilizar esta programação em tempo de resposta aceitável.

Será demonstrada nesta dissertação a facilidade e eficiência em disponibilizar uma interface *web* para que o usuário possa inserir simples parâmetros do problema JSP

através do seu navegador *web* e obter, em tempo real, o escalonamento das tarefas deste problema.

Para cada solução viável encontrada para o problema “*postado*” é disponibilizado um arquivo XML, podendo também esta solução ser visualizada através de um documento PDF, com a apresentação de um Gráfico de *Gantt* e de *linha XY*, descrevendo detalhadamente, assim, esta solução viável para a análise e tomada de decisão do pesquisador.

O Simulador desenvolvido é estruturado para executar as diversas heurísticas ou metas-heurísticas conhecidas na literatura, de forma que se torne possível estudar a implementação destes em uma arquitetura serial/paralela e, assim, estes pesquisadores e engenheiros podem adotar o método que melhor se adequa às necessidades da indústria.

No presente trabalho, um método heurístico de otimização foi também utilizado como estudo de caso e inspirado na *Busca Tabu* de Taillard (1994). A heurística implementada no Simulador tem como objetivo utilizar núcleos, das centenas de núcleos disponíveis da GPU, para processar *threads*. Cada *thread* representa e processa uma das operações a serem escalonadas para o problema (JSP). Cooperativamente, essas *threads* trocam informações com as outras *threads* de modo a encontrar uma solução viável. De todas as estimativas da população de *threads* é escolhida a melhor solução para o problema (JSP).

1.5 Organização

Este trabalho é composto de sete capítulos organizados da seguinte forma:

O **Capítulo 1** apresenta uma introdução ao problema JSP, a dificuldade encontrada na sua resolução, sua aplicabilidade industrial, os objetivos e contribuições do presente trabalho na resolução deste difícil problema combinatorial.

Apresenta-se no **Capítulo 2** uma introdução e fundamentação básica conceitual do problema JSP, através de uma breve descrição do estado da arte com as principais técnicas ou métodos desenvolvidos para a resolução deste problema, as descrições dos gráficos utilizados nesta dissertação, como o grafo Disjuntivo, gráfico de Gantt e

o gráfico de linha XY (Evolução), os quais representam o problema no seu aspecto inicial e final, respectivamente.

No **Capítulo 3** é apresentado o algoritmo, como um estudo de caso, com a descrição de uma heurística paralela adotada para a elaboração de uma solução viável ao problema JSP, baseada na *Busca Tabu* de Taillard (1994), utilizando o potencial da arquitetura paralela existente no Simulador.

O **Capítulo 4** é dedicado ao fornecimento de uma visão geral do que se constitui uma definição de computação paralela, os modelos de computação paralela, desempenho de computadores paralelos, a arquitetura da série G80 da NVIDIA, sua estrutura e recursos, a arquitetura de software CUDA. Por fim, exemplos de uso da arquitetura paralela utilizando CUDA da NVIDIA.

O **Capítulo 5** apresenta a arquitetura do ambiente do Simulador utilizado para avaliar o algoritmo com a descrição dos módulos tecnologia *web*, o qual é implementado utilizando a plataforma Java (J2EE), descrição do modelo MVC (“*model-view-controller*”), camada da interface JNA e a camada C que implementa o algoritmo (*kernel*) no dispositivo ou GPU e funções de chamadas do *kernel* no computador *Host* ou CPU, são apresentadas as interfaces de usuário e seus relacionamentos com as informações do modelo JSP.

No **Capítulo 6** são apresentados os experimentos e resultados do modelo computacional paralelo abordado no Simulador para oito instâncias clássicas, ou *bench-mark*, disponíveis na *OR-Library*.

O trabalho completa-se com o **Capítulo 7** descrevendo as considerações finais e as propostas para novas investigações no tema.

CAPÍTULO 2

Fundamentação Teórica

2.1 Definição do Problema “Job Shop”

No modelo clássico de escalonamento de tarefas ou “*Job Shop*”, é possível descrever o problema como um conjunto finito de tarefas J (*jobs*) onde estas são processadas por um conjunto finito de máquinas ou robôs R . (SEDA, 2007). Cada tarefa consiste de uma seqüência de operações, pertencentes ao conjunto finito de operações O . Estas serão executadas pelos robôs, sem interrupção, por um período de tempo previamente definido. Cada robô pode executar uma operação por vez. Porém, uma tarefa pode ser executada por todos os robôs e as operações sucessivas da mesma tarefa poderão ser executadas por diferentes robôs.

Portanto, o escalonamento é a designação das operações, no intervalo do tempo, para serem executadas pelos robôs.

Haverá um conjunto de soluções do problema que é exponencial no número de robôs. Assim, a resolução deste problema é atingir o objetivo em encontrar um escalonamento com o menor tempo para execução de todas as tarefas.

Seja a seguinte notação:

- Para cada tarefa $j \in J$, teremos $j = \{O_1, O_2, \dots, O_m\}$, onde o número total de tarefas é $n = |J|$;
- O conjunto de robôs $r \in R$, onde $r = \{1, 2, \dots, m\}$ e m é o número total de robôs;
- O conjunto de operações $o \in O$, onde $|O| = n \times m$ são as operações realizadas pelos m robôs em cada das n tarefas ;
- J_i - tarefa na qual a operação i pertence;
- R_i - robô no qual a operação i será executada;
- t_i - tempo inicial para executar a operação i ;
- p_i - tempo (peso predefinido) para executar a operação i ;

- C_{max} - quantidade de tempo para concluir o conjunto das tarefas, a qual é a função objetivo chamada *makespan*, onde:

$C_{max} = \max[C_j], \forall j; 1 \leq j \leq n$, sendo que C_j é o maior tempo para terminar a última operação executada da tarefa j .

No referido conjunto de operações O , a relação binária $\rightarrow$ define uma regra de precedência entre as operações de uma mesma tarefa. Por exemplo, se $i \rightarrow j$ a operação i é predecessora da operação j e pelas especificações do problema JSP, descritas acima, $R_i \neq R_j$. Esta restrição é uma restrição temporal, o que certifica que a seqüência de processamento das operações em cada tarefa corresponde a uma ordem predeterminada.

Para obter a melhor solução do problema JSP, encontra-se o tempo inicial t_i para cada operação $i \in O$ tal que:

$$\max(t_i + p_i), \forall i \in O \Rightarrow t_i \geq 0 \text{ seja minimizado}$$

Sujeito as seguintes restrições:

- $\forall i, j \in O, i \rightarrow j : t_j \geq t_i + p_i$
- $\forall i, j \in O, i \neq j, R_i = R_j \Rightarrow (t_j \geq t_i + p_i) \vee (t_j \geq t_i + p_j)$

As restrições acima são de recurso, ou seja, apenas uma *operação* será atendida por um *robô* em um determinado momento. Qualquer escalonamento que atenda essas restrições é chamada de escalonamento de tarefas “*job Shop*” ou solução viável.

2.2 Estado da Arte

No mundo globalizado, destaca-se o setor industrial onde a competitividade e produtividade estão diretamente relacionadas ao controle e gerenciamento da produção (CHONG *et al.*, 2006). Com a intensa competição do comércio global, que resultou no desafio da fabricação de produtos a baixo custo, em ciclos de tempo de produção cada vez mais curtos e para uma crescente variedade de produtos, a

indústria deparou-se com problemas de programação que surgem quando diversos recursos comuns (equipamentos, tarefas, trabalhadores, matéria-prima, etc...) devem ser distribuídos durante um mesmo período de tempo para a fabricação de diversos produtos.

A programação da produção pode ser definida como o escalonamento e planejamento no tempo para a execução de uma sequência de ordens no processo de produção de um produto (PINEDO, 1995). Esta programação é realizada através de uma série de operações em um conjunto de recursos disponíveis. Como resultado, tem-se um programa de produção que especifica as atribuições temporais das operações das ordens nos recursos a serem utilizados.

Esta demanda crescente para o controle e planejamento da produção fez com que diversos pesquisadores mantivessem esforços na resolução do problema apresentado nesta dissertação, escalonamento de tarefas (JSP), viabilizando, desta forma, sistemas para o controle da produção industrial.

A dificuldade não está somente no fato do problema (JSP) ser considerado um problema NP-Difícil, mas também por possuir a reputação de ser um dos maiores problemas de otimização combinatória (CHAO YONG *et al.*, 2008). Esta característica de inflexibilidade fez com que este problema tenha sido um dos mais estudados desde a década de 50, e isso explica o constante desenvolvimento de novas técnicas que propõem abordagens mais eficientes e escopo de soluções mais robustas, especialmente vistas na década de 90.

Nas primeiras soluções foram utilizados métodos exatos e apesar da sua importância em obter uma solução ótima, estas ficam restritas a pequenos problemas pelo esforço computacional exigido para completar a execução do método.

Com o rápido progresso da tecnologia da informação foi possível encontrar soluções para problemas em maior escala. Métodos ditos aproximativos foram desenvolvidos para problemas em larga escala (HARUN *et al.*, 2008). Apesar destes métodos *aproximativos* não garantirem uma solução ótima eles garantem uma solução sub-ótima, que são perfeitamente aceitáveis.

Para que a programação da produção industrial seja executada de forma eficaz é necessário que fique disponível para a indústria métodos computacionais robustos, eficientes e que possuam um alto poder de processamento, de forma a obter resultados rápidos e precisos, pois a escala dos problemas de programação está aumentando.

Os métodos encontrados no estado da arte para a programação da produção foram tradicionalmente processados em computadores com processadores de um núcleo, ou “*single-core*”, por questões de restrição tecnológica. Os *softwares* utilizados na programação destes computadores baseiam-se em algoritmos seriais e estes têm encontrado dificuldades para aumentar o seu desempenho: o “*single-core*” chegou ao seu limite tecnológico, como será visto no Capítulo 3.

Com o requisito do alto desempenho, poderia ser adotado o uso de estações de trabalho paralelo de grande porte, mas estas em geral são proprietárias, e devido ao seu custo se tornam inacessíveis para a maior parte dos usuários.

Outra opção que poderia ser utilizada para fazer uso do potencial de uma arquitetura paralela seria a adoção de diversos computadores “*desktop*” de menor custo, interligados através de uma rede de interconexão, compondo desta forma um “*cluster*” de computadores no modelo de arquitetura MIMD: cada máquina possui um *software* próprio e, assim, a melhor solução para o problema de programação ou escalonamento seria escolhida dentre todas as soluções encontradas. Entretanto, esta arquitetura necessita de compiladores específicos que façam uso de uma Interface de Passagem de Mensagem (MPI) disponíveis em bibliotecas de funções para as linguagens C ou Fortran, assim, há a dificuldade na troca de informações assíncronas entre cada solução, pois dependem de uma rede de interconexão de baixa latência.

Existem também, *softwares* de simulação para o escalonamento que foram desenvolvidos nos últimos anos pelos meios acadêmicos ou pela indústria, os quais são de possível acesso por apenas alguns sistemas acadêmicos que são distribuídos livremente na *internet*, como exemplo:

- Legin¹ é um software de simulação para o escalonamento desenvolvido pela *Stern School of Business* como ferramenta de aprendizagem para o escalonamento de ambientes de produção para máquinas paralelas ou para apenas uma única máquina do problema *flow shop*, *flexible flow shop* e *flexible job shop*. O sistema é um ambiente gráfico utilizando uma máquina *desktop Windows'95* ou superior. Possui heurísticas e algoritmos de escalonamento próprio e permite que o usuário teste suas heurísticas com as do sistema. Foram identificadas limitações nesta solução como ambiente individual e mono processado e não disponibiliza os recursos computacionais em ambiente compartilhado, como um laboratório.
- Applet² apresentado pela Universidade de Bern possibilita definir até 5 ordens de produção ou *jobs*. Cada *job* possui 5 *operações* para serem processados em 5 *máquinas*. As seqüências de *operações* podem ser trocadas arrastando e soltando a operação escolhida pelo usuário. Esta aplicação é meramente didática e não possibilita o desenvolvimento de novas heurísticas, além das restrições impostas pela tecnologia *Java-Applet*.

2.3 Métodos de resolução

Problemas de otimização buscam a melhor solução de acordo com a função objetivo, entre as muitas soluções possíveis. No presente trabalho busca-se o menor *makespan* do problema JSP. Vários são os possíveis métodos de resolução para o problema de escalonamento de tarefas e a seguir, são apresentados os mais relevantes.

2.3.1 Métodos Exatos

O uso de um método exato e determinístico para versão simples do problema JSP é considerado difícil para que seja encontrada uma solução ótima global, e é computacionalmente intratável para problemas de grande dimensão.

Algoritmos que usam métodos exatos obtêm a solução ótima através de regras simples aplicadas sobre o conjunto que contém todas as soluções. Abordagens para problemas específicos e consideração de critérios atenuam o tamanho do problema,

¹ – LEKIN “Flexible Job-Shop Scheduling System” <http://www.stern.nyu.edu/om/software/lekin/>

² – Criado por Christoph Burkhalter <http://www.ie.iwi.unibe.ch/forschung/e-learning/lernobjekte/resource/seqplanning/applet/jobShop.html>

diminuindo o impacto causado pela dimensão do conjunto de soluções já que se aproveitam de particularidades deste problema. Esses algoritmos têm seu tempo computacional variando polinomialmente com o tamanho do problema. Obviamente, este tipo de algoritmo é aplicável apenas para pequenos problemas, onde o tratamento computacional possa ser viável. Encontra-se na literatura: Johnson (1954), Giffler e Thompson (1960), Carlier e Pinson (1989), Applegate e Cook (1991), Gray e Hoesada (1991), Brucker *et al.* (1994). Bons resultados foram encontrados em pequenas instâncias como a 10x10 publicada no trabalho de Fisher e Thompson (1963). Instâncias do tipo 25x15 já são consideradas intratáveis pelo método exato.

2.3.2 Métodos aproximativos

Métodos ditos aproximativos foram desenvolvidos pela necessidade de diminuir o enorme tempo computacional necessário para a resolução de grandes problemas JSP. Estes métodos eficientes diminuiriam muito o esforço computacional destes problemas classificados como “*NP- difíceis*”, mas estes não garantem uma solução ótima, mas apenas uma solução sub-ótima do problema.

2.3.2.1 Heurísticas

Para a resolução de problemas intratáveis computacionalmente, se faz necessária a aplicação de heurísticas. Heurísticas são regras que pretendem obter uma aproximação da solução de forma intuitiva, neste método a estrutura do problema deve ser interpretada e explorada inteligentemente para obtenção de uma solução razoável (NICHOLSON, 1971).

Análises de métodos heurísticos para *Job Shop* foram feitas por Pinson *et al.* (1995), Vaessens *et al.* (1996) e Cheng *et al.* (1999). Algoritmos de busca local por Lourenço (1995) e Vaessens *et al.* (1996).

2.3.2.2 Metas-Heurísticas

Uma meta-heurística é um método heurístico para resolver de forma genérica problemas de otimização. Até o momento, são técnicas que encontraram as melhores soluções para o “*Job Shop*” em tempo polinomial. Existem várias técnicas que obtiveram sucesso com a resolução do problema JSP, como recozimento

simulado ou “*Simulated Annealing*” (SA), algoritmo genético ou “*Genetic Algorithm*” (GA) e busca tabu ou “*Tabu search*” (TS).

Outra classe de meta-heurística é a inteligência coletiva, a qual é baseada no comportamento de insetos sociais, como as formigas e abelhas, as quais prosperaram nos últimos 140 milhões de anos. Sua característica principal é que estes insetos são individualmente limitados para resolver problemas, possuem comportamentos individuais limitados e previsíveis na execução de suas tarefas, são em sua maioria muito bem definidas. Porém, coletivamente, uma colônia ou enxame é capaz de resolver problemas complexos, tais como encontrar o menor caminho entre a colônia e o alimento, formação de pontes (formigas), migração maciça da colônia, exploração das melhores fontes de alimentos, regulação de temperatura na colméia ou ninho etc.

Muitos problemas de otimização multi-variável não podem ser resolvidos exatamente em tempo polinomial. Isto gera muito interesse pela comunidade acadêmica em desenvolver algoritmos de busca que tenham soluções perto da ótima, em tempos razoáveis. Otimização baseada em enxame ou “*Swarm Optimization Algorithm*” (SOA) imita os métodos encontrados na natureza para direcionar os métodos de busca em direção a solução ótima.

Técnicas de Inteligência de Enxame ou “*Swarm Intelligence*” (SI) foram inicialmente criadas por Beni e Wang (1990) na área da robótica (MARTIJN, 2007) e posteriormente estendidas por Marco Dorigo (DORIGO e STÜTZLE, 2000), criador do ACO (“*Ant colony Optimisation*”), uma meta-heurística para problemas de otimização combinatória. Esta técnica baseia-se na interação de muitos agentes simples que tentam atingir um objetivo específico, ou uma solução para problema complexo.

Desta forma, algoritmos de otimização robustos e sistemas adaptativos foram desenvolvidos através de modelos computacionais inspirados nestes mecanismos evolutivos. Como exemplo, temos o uso da técnica ACO das formigas forrageiras, que é empregado como solução ao problema do caixeiro viajante (“*Traveling Salesman Problem*”) ou TSP (um problema NP-Completo) que em um tempo de processamento razoável, obtém uma solução viável, perto da solução ótima.

2.4 Instâncias de problemas JSP

Instâncias de problemas JSP foram idealizadas para avaliar a evolução dos algoritmos criados por diversos autores para este tipo de problema. Estas instâncias são conjuntos de dados predefinidos que perfazem um “*benchmark*” de problemas, com diversos níveis de dificuldade. Por exemplo, uma instância específica do problema formulada e apresentada no livro de Muth and Thompson (APPLEGATE e COOK, 1991) no ano de 1963, ficou 20 anos sem solução. Esta instância consistia de um problema JSP para 10 máquinas por 10 tarefas, e ela somente foi resolvida em 1985 por Carlier e Pinson. Este problema ficou conhecido como *mt10*. Estes autores formularam outras duas instâncias que se tornaram os “benchmark” *mt06* e *mt20*. Estas instâncias estão disponíveis na OR-Library³.

Outras instâncias de problemas JSP apresentadas são o *abz5* e *abz6*. Estes são dois problemas propostos por Adams, Balas e Zawack; os problemas *lab19* e *lab20* representam instâncias formuladas por Lawrence. A instância de problemas *orb1* até *orb5* foram propostas no ano de 1986, e as sequências das operações para cada problema foram criadas por convidados do autor (com o desafio de elaborarem “problemas difíceis”). Para cada problema eram gerados dois conjuntos de tempos de processamento aleatoriamente.

2.5 Grafo Disjuntivo

O grafo disjuntivo é um dos modelos mais populares para descrever um problema de escalonamento JSP (BLAZEWICZ *et al.*, 2000) e a maioria dos métodos desenvolvidos para a solução do problema JSP, exatos ou aproximativos, pode ser representada por este modelo de grafo originalmente proposto por Roy e Sussmann (1964).

Através de um grafo, são modeladas todas as informações ordenadas para descrever uma solução parcial ou completa de um problema JSP. Assim, a própria representação ajuda significativamente na resolução do problema.

O conjunto de possíveis soluções para o problema JSP pode ser representado por este grafo: arcos conjuntivos – representa a relação de precedência entre operações

³ OR-Library Brunel University – West London -.URL: <http://people.brunel.ac.uk/~mastijb/jeb/orlib/files/jobshop1.txt>

das tarefas; arcos disjuntivos – representa as possíveis relações de precedência entre as operações a serem executadas pela mesma máquina ou robô.

Por definição, o grafo Disjuntivo é um conjunto de nós (vértices) e arestas (arcos). Os nós são ligados pelas arestas do grafo. Assim um grafo disjuntivo G é definido como:

$$G = (O, E_{conj} \cup E_{disj})$$

- Vértices O representam o conjunto das operações a serem escalonadas.
- Arcos E_{conj} representam um conjunto de arcos direcionados que demonstram as regras de precedência para cada tarefa.
- Arcos E_{disj} representam um conjunto de arcos disjuntos bidirecionais, que demonstram todas as operações que precisam ser manipuladas por uma máquina ou robô específico.
- Para cada arco do grafo disjuntivo G , são associados custos que correspondem ao tempo de processamento para cada operação.
- Dois vértices especiais não fazem parte do problema e representam as operações iniciais e finais das operações do problema e possuem custo zero.

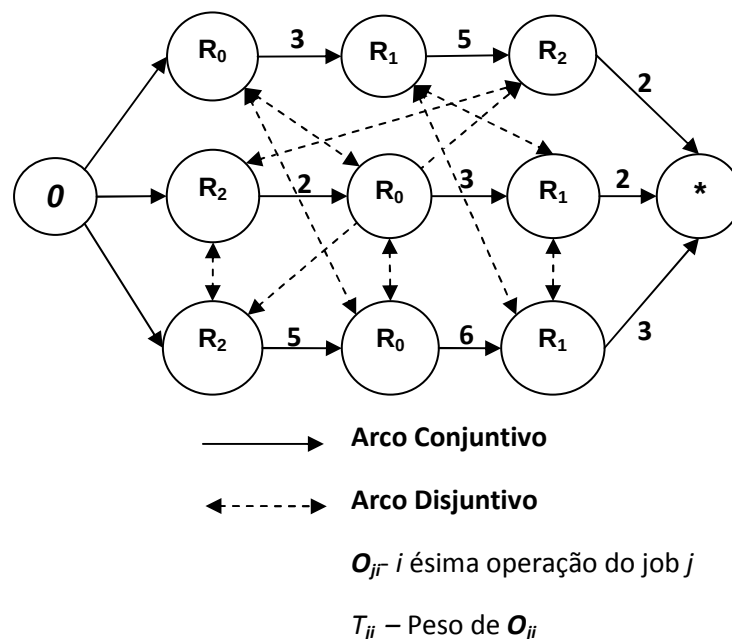


Figura 1: Representação de um problema JSP através de Grafo Disjuntivo.

Um exemplo de um grafo disjuntivo para um problema JSP pode ser visto na Figura 1, onde este é baseado em três tarefas para três robôs.

Na Tabela 1, cada linha representa uma ordem de precedência de tarefas distribuídas entre as máquinas do conjunto $R \{0,1,2,...,n-1\}$ e, também, a representação de seus pesos (tempos) entre parênteses.

Tabela 1: Exemplo para a ordem de operações e duração de operações para cada tarefa.

Operação	0	1	2
Tarefa			
0	$R_0 (3)$	$R_1 (5)$	$R_2 (2)$
1	$R_2 (2)$	$R_0 (3)$	$R_1 (2)$
2	$R_2 (5)$	$R_0 (6)$	$R_1 (3)$

Uma solução viável pode ser obtida convertendo os arcos disjuntos bidirecionais em arcos direcionados. Deste modo uma possível solução é obtida assegurando-se que os ciclos existentes no grafo sejam desfeitos. Isto tornará o grafo disjuntivo em um grafo direcionado como mostrado na Figura 2.

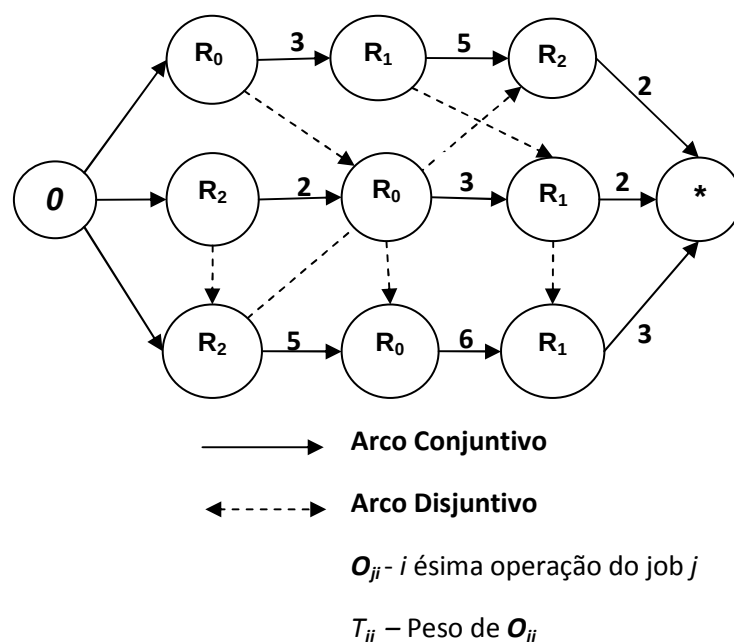


Figura 2: Uma solução viável para o problema JSP definido na Tabela 1.

A complexidade do problema JSP aumenta com o número de robôs e o número de tarefas. O tamanho do espaço de busca (KAAZI e NAWAZ, 2007) é $(n!)^m$, isto é, o resultado do fatorial de tarefas aumenta exponencialmente com o número de robôs. Na prática, o problema JSP do mundo real possui grande número de tarefas e robôs, assim como regras adicionais, o que aumenta em muito sua complexidade.

2.6 Gráfico de Gantt

Para visualizar a solução do problema JSP, o escalonamento ou *scheduling* é projetado em um *Gráfico de Gantt*⁴. Yamada e Nakano (1997) descrevem esta forma como a mais conveniente para representar uma solução JSP. O gráfico mostra as unidades de tempo na abscissa e o escalonamento das máquinas no eixo das ordenadas detalhando as informações das operações escalonadas para cada máquina. Por exemplo, é apresentada uma solução viável para o problema de escalonamento descrito na Tabela 1 através do gráfico clássico de *Gantt* (Figura 3): os blocos representam as operações e o comprimento do bloco representa o seu custo.

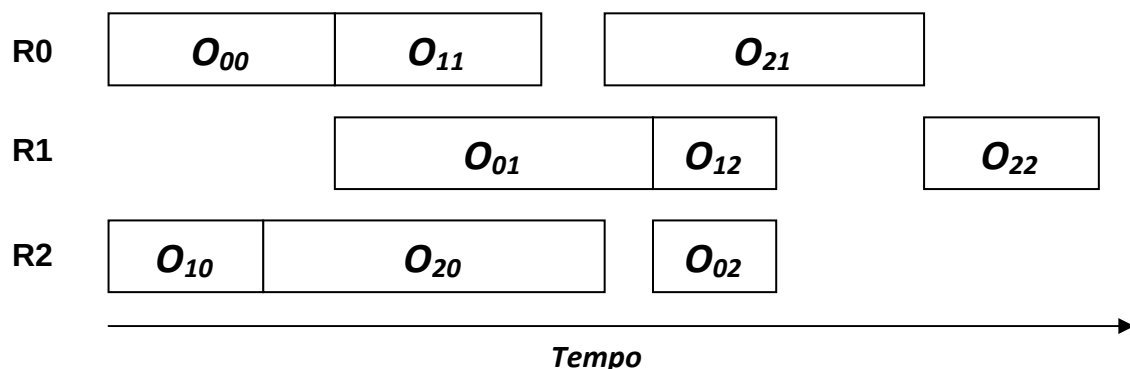


Figura 3: Representação da seqüência de escalonamento das operações de cada tarefa, para as máquinas definidas na Tabela 1, demonstrando uma das possíveis soluções viáveis ao problema JSP.

Um exemplo de um *Gráfico de Gantt* gerado pelo Simulador da presente dissertação, a ser apresentado no Capítulo 5, é mostrado na Figura 4 correspondendo às informações do problema três x três descrito no exemplo da

⁴ Wikipedia – “Desenvolvido em 1917 pelo engenheiro social Henry Gantt, esse gráfico é utilizado como uma ferramenta de controle de produção. Nele podem ser visualizadas as tarefas de cada membro de uma equipe, bem como o tempo utilizado para cumpri-la. Assim, pode-se analisar o empenho de cada membro no grupo.

Tabela 1 e visto através da versão clássica do gráfico na Figura 3. Nesta figura, cada barra representa as operações O_{ji} , onde j é a tarefa e i é a operação. O eixo x é a coordenada horizontal cujo comprimento representa o tempo de processamento t_{ji} . O tempo máximo para executar todas as O_{ji} operações de escalonamento é definido como função objetivo do problema. No exemplo em questão o *makespan* é 16.

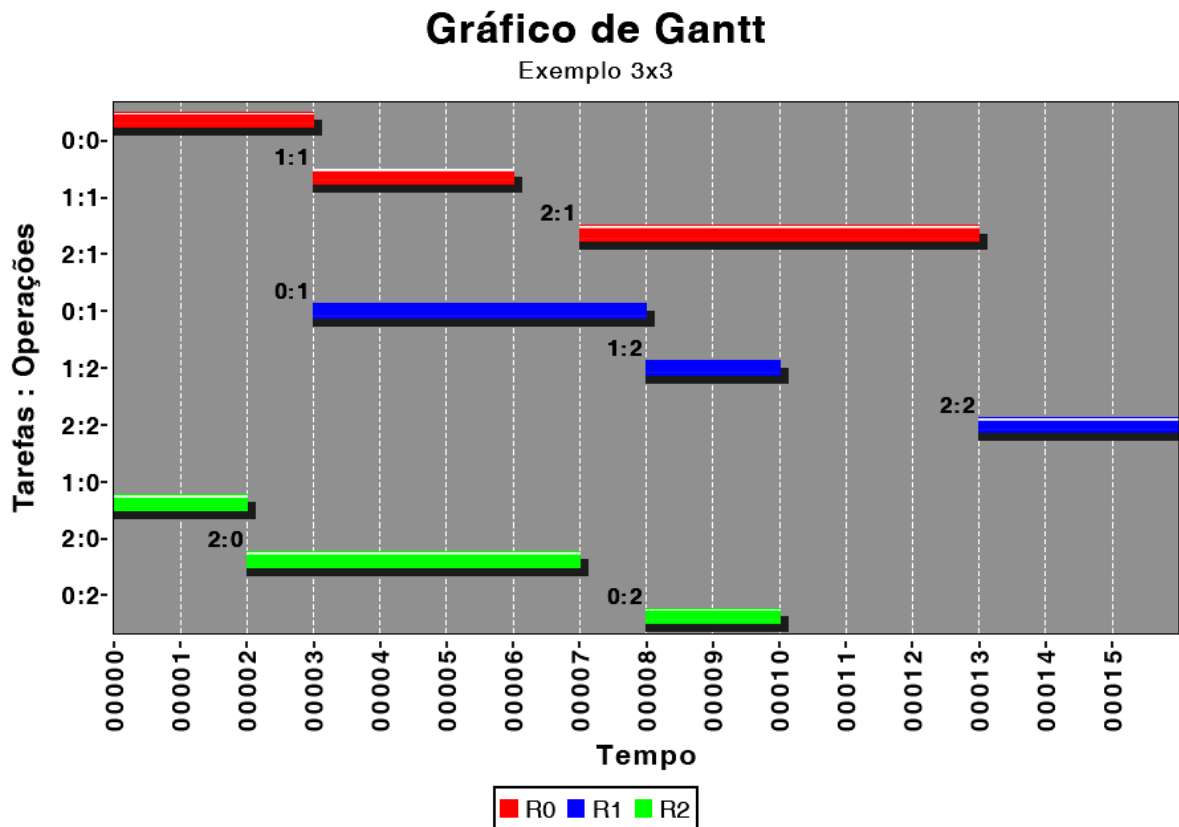


Figura 4: Gráfico de Gantt gerado pelo simulador da presente dissertação, descrevendo outra solução viável ao problema JSP definido na Tabela 1.

2.7 Gráfico XY - Evolução

A forma de escrever o algoritmo para arquitetura paralela é fundamental para o ganho de desempenho e qualidade da solução, e é diferente de escrever o mesmo algoritmo para um processamento serial.

Ao desenvolver o algoritmo o pesquisador poderá utilizar um gráfico de evoluções inserido no documento PDF gerado pelo Simulador para acompanhar a eficiência ou problemas no código desenvolvido, como por exemplo, ciclagem ou dependência de dados, ótimos locais, diversificação e intensificação da busca local no espaço de busca.

Evolução

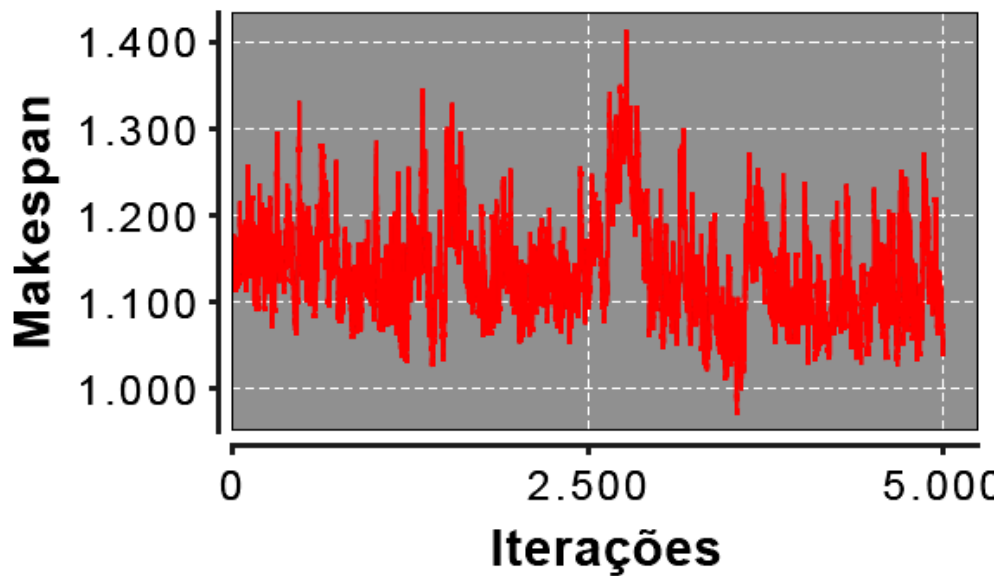


Figura 5: Gráfico de Evolução gerado pelo simulador para o problema MT10.

O gráfico (Figura 5) mostra o comportamento do algoritmo no processo de busca de soluções viáveis. Para cada solução encontrada é calculado seu *makespan*. Para a heurística que estudamos, mesmo quando uma solução encontrada é apenas um ótimo local, a busca continua em outras regiões do espaço de busca perseguindo o ótimo global até um número pré-determinado de iterações.

Capítulo 3

Algoritmo - Busca Tabu Paralelo

3.1 Introdução

Qual método poderia ser mais apropriado na solução do problema JSP em um ambiente de arquitetura paralela?

Como visto no Capítulo 1, são utilizados modelos de busca exatos como o “*Branch and Bound*” para encontrarmos uma solução ótima com o objetivo de resolver o problema JSP. Entretanto, é necessário um grande esforço computacional e, dependendo do tamanho do problema, o mesmo é intratável computacionalmente.

No ambiente de produção real, os problemas não são pequenos. A otimalidade não é atendida em tempo computacional satisfatório. Como visto no Capítulo anterior, os métodos aproximativos demonstraram ser mais eficientes com resultados próximos do ótimo, e estes podem ser aceitos para grandes problemas, pois o referido tempo computacional é mais satisfatório.

Nosso estudo se deteve nos algoritmos aproximativos, procurando minimizar o tempo de resposta do Simulador, de modo que fosse viável tecnicamente produzir uma resposta sem haver interrupção, por exemplo, na conexão por *timeout* entre o cliente e o servidor *web*.

Uma das técnicas usadas foi a adoção de uma estratégia de escalonamento utilizando uma arquitetura paralela de baixo custo buscando reduzir o tempo de execução para a solução do problema (JSP) partindo do conhecimento do caminho crítico da solução corrente, ou seja, da maior seqüência de dependências de dados entre tarefas.

O método heurístico *Busca Tabu* de Taillard (1994), é uma versão da estratégia descrita acima. Ele é baseado no uso de estruturas de memória de forma a fugir da convergência para um ótimo local. Procurou-se testar este método na estrutura de paralelismo, utilizada pelo Simulador, como um estudo de caso da solução paralela do problema JSP.

Na organização do processamento do Simulador, um conjunto de soluções é estimado a cada iteração. Cada vértice do grafo disjuntivo, apresentado na Seção 2.5, é representado por uma *thread* do processamento paralelo e esta é escalonada na GPU concomitantemente com as outras *threads*: estas últimas processam as operações das tarefas e as arestas (disjuntivas e conjuntivas) representam a dependência de dados (ou comunicação) entre as operações das tarefas.

A estratégia escolhida é naturalmente adequada e integrada a uma arquitetura paralela contendo centenas de processadores, onde cada um destes pode estimar uma solução e esta ser escolhida de acordo com a aptidão ou “*fitness*”, ou seja, o tempo máximo para concluir todas as tarefas ou o *makespan* da solução.

3.2 Motivação

“Junto com *recozimento simulado* (SA) e algoritmo *genético* (GA), a *Busca Tabu* (TS) foi amplamente reconhecida pelo *Committee on Next Decade of Operations Research* (CONDOR, 1988) como extremamente promissora para o tratamento futuro de aplicações práticas” (GLOVER e LAGUNA, 1997).

A previsão descrita acima continua atual: “dentro dos métodos metas-heurísticas, a *Busca Tabu* é um dos mais promissores para realizar o escalonamento de um problema JSP utilizando como critério o *makespan*” (CHAO *et al.*, 2008).

Com esta motivação, foi desenvolvido um método TS baseado na heurística computacional de busca, conhecido por geralmente transpor o problema de convergência local de forma eficiente.

Utilizando os diversos recursos computacionais de uma arquitetura paralela SIMD, disponibilizada na GPU, a qual é o elemento principal da arquitetura do Simulador, apresentado nesta dissertação, para processar o método de *Busca Tabu* (TS) no seu núcleo (*kernel*), a ser visto no Capítulo 4, obtendo-se, assim, a descentralização das ações do escalonamento com alta interação individual entre as *threads*, onde cada *thread* persegue a melhor solução.

A heurística desenvolvida para a solução do problema (JSP) procura utilizar a referida multiplicidade de *threads* processadas em um núcleo da GPU. Cada núcleo, das centenas de núcleos disponíveis no multiprocessador, processa *threads* que

procuram a melhor solução para o problema, e suas informações são compartilhadas colaborativamente entre estas de forma a obter a melhor solução do conjunto de soluções, ou o *makespan* do problema JSP.

3.3 Busca Tabu (TS)

Busca Tabu ou “Tabu Search” (TS) foi proposto por Fred Glover (GLOVER, 1986). Esta meta-heurística utiliza o método de busca local e explora o espaço de soluções movendo-se de uma solução para outra, escolhendo seu melhor vizinho. Para isso, uma estrutura de dados armazena as características das soluções encontradas e possibilita a TS escapar do ótimo local.

A *Busca Tabu* possui três princípios fundamentais: (GLOVER e LAGUNA, 1997):

- O uso de uma estrutura de memória para guardar o histórico da evolução no processo de busca;
- O uso de um método de controle para tomar a decisão entre a aceitação ou não de uma nova configuração, com base nas informações registradas na *lista tabu* referentes às restrições e aspirações desejadas;
- A incorporação de procedimentos que alternam as estratégias de intensificação e diversificação.

A estratégia em diversificar é uma tentativa realizada pelo algoritmo para buscar em outras regiões do espaço de busca uma nova solução, evitando que esta busca pare na primeira solução ótima local. A outra estratégia é a intensificação onde o algoritmo procura explorar o espaço de busca na vizinhança da solução corrente na tentativa de encontrar a melhor solução.

3.4 Busca Tabu Paralela

Pioneiramente, a *Busca Tabu* foi implementada ao problema JSP por Taillard (CHAO *et al.*, 2008), onde este utilizou a estrutura de busca pela vizinhança descrita por Van Laarhoven *et al.* (1992). O algoritmo TS paralelo, utilizado como estudo de caso no Simulador, foi inspirado na técnica descrita no artigo de Taillard (1994) onde é apresentada uma estratégia de movimentos estimados para orientar a busca local, e será descrito posteriormente.

3.4.1 Heurística Paralela

No sistema desenvolvido, os escalonamentos (JSP) são de natureza paralela: as máquinas ou robôs que trabalham em paralelo no tempo para a execução de suas operações.

Foi visto no Capítulo 2 que no modelo matemático do problema (JSP), a primeira restrição é uma restrição temporal, o que certifica que a seqüência de processamento das operações em cada tarefa corresponde a uma ordem predeterminada. A segunda restrição significa que há uma restrição de recurso onde apenas uma operação será atendida por uma máquina em um determinado momento e, a terceira restrição certifica a execução de todas as operações.

Lembrando a definição do problema descrita na Seção 2.1, o problema JSP é definido como sendo um conjunto J de n tarefas e um conjunto de robôs $r \in R$. Cada tarefa $j \in J$ possui m operações e cada operação $O_{j,i}$, onde j indica a tarefa a qual a operação pertence e i a ordem desta operação na tarefa, deverá ser escalonada em um robô previamente definido, por um tempo fixo também previamente definido, sem permitir interrupções na execução desta operação. Nenhum robô deverá executar mais que uma operação por vez e cada operação $O_{j,i}$ deverá estar concluída antes que a próxima operação $O_{j,(i+1)}$ comece a ser executada.

Como descrito no referido Capítulo 2, convertendo-se os arcos disjuntivos bidirecionais em arcos direcionados obtém-se uma solução viável. Se removidos os arcos disjuntivos redundantes, toda operação terá exatamente dois sucessores e dois predecessores sendo um relativo à tarefa vizinha e outro relativo à máquina na qual esta operação está sendo executada. Ficaria apenas faltando o cálculo dos tempos iniciais e finais para cada operação do problema (JSP).

A seguinte notação será utilizada para definir alguns relacionamentos entre as operações em escalonamento para o problema JSP.

Seja:

- $TS_{[a]}$ a operação sucessora de a pertencente na mesma tarefa da operação a .

- **RS_[a]** a operação sucessora de ***a*** cuja execução é feita pelo mesmo robô que executa ***a***.
- **TA_[a]** a operação antecessora de ***a*** pertencente na mesma tarefa da operação ***a***.
- **RA_[a]**: a operação antecessora de ***a*** cuja execução é feita pelo mesmo robô que executa ***a***.

Todas as operações possuem um tempo inicial de execução denotado por t_i (“*release time*”) e um tempo máximo para atingir a operação final.

No caso da operação i pertencer ao caminho crítico, que será descrito na Seção 3.4.2, utiliza-se a seguinte identidade:

$$C_{max} = c_i + t_i .$$

No algoritmo paralelo, cada *thread* processa o tempo inicial t_i e o tempo final t_f da operação em conjunto com as outras *threads* representativas no grafo para a solução, utilizando o seguinte método para calcular o tempo inicial e final de cada operação de maneira construtiva:

$$t_{i[a]} = \max(t_{TA[a]} + p_{TA[a]}, t_{RA[a]} + p_{RA[a]})$$

$$t_{f[a]} = t_{i[a]} + p_{i[a]}.$$

Estas informações são armazenadas por cada *thread* para computar o escalonamento de uma solução ***S*** e são processadas em paralelo na ordem de número de operações. Desta forma, a abordagem utilizada pelo algoritmo é escalonar em paralelo cada solução ***S***, no tempo, a cada iteração por todas as *threads* em conjunto, definindo os tempos iniciais e finais de cada operação. Estas informações são armazenadas em uma memória global do dispositivo GPU de forma que qualquer *thread* tenha acesso às informações das *threads* vizinhas:

$$t_{TA[a]}, p_{TA[a]}, t_{RA[a]} \text{ e } p_{RA[a]}$$

Por exemplo: no grafo ***G*** (Seção 2.5) a *thread*₂, que processa o nó ***O*₀₂**, poderá ser escalonada quando os tempos finais das operações predecessoras estiverem

disponíveis tanto a operação predecessora pertencente à mesma tarefa quanto operação pertencente ao mesmo robô, como mostrado na Figura 6.

O tempo de saída resultante do processamento da $thread_2$ será a soma do seu peso programado (2) para esta operação adicionada ao maior dos tempos finais entre seus vizinhos. No exemplo, o tempo final resultante é 10.

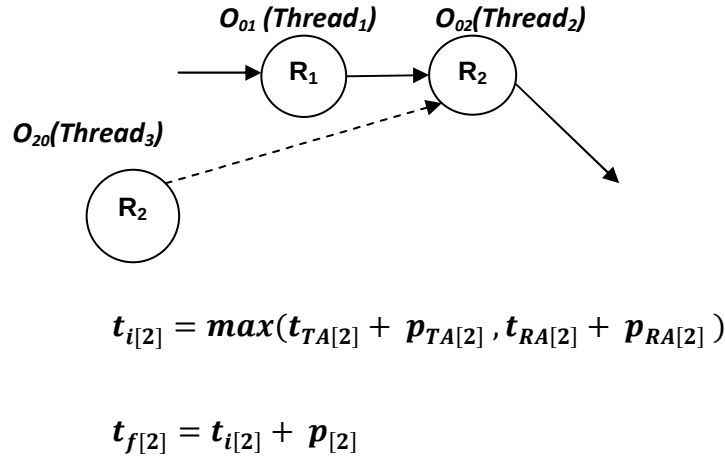


Figura 6: Identificação do tempo inicial e final, por uma *thread* que representa um nó do grafo **G**.

Algoritmo 1 – Cálculo para o tempo inicial (release time) em cada operação.

```

Seja  $Q = \{i \mid RA_{[i]} \text{ e } TA_{[i]} \text{ não são definidos}\};$ 
inicializar a variável booleana  $sinc_i$  para  $thread_i$  como false;
enquanto  $Q \neq \emptyset$  faça
    se  $t_i \in Q$  então
         $Q \leftarrow Q \setminus \{i\}$  para  $thread_i$ 
         $sinc_i \leftarrow \text{true};$ 
        Calcula  $t_i = \max(t_{TA} + p_{TA}, t_{RA} + p_{RA})$ 
        Se  $sinc_i$  de  $RA_{TSi}$  é verdadeiro então;
             $Q \leftarrow Q \cup \{TS_i\}$  para  $thread_{TS[i]}$ ;
        fim se
        Se  $sinc_i$  de  $TA_{RSi}$  é verdadeiro então;
             $Q \leftarrow Q \cup \{RS_i\}$   $thread_{RS[i]}$ ;
        fim se
    fim se
fim enquanto

```

A *Busca Tabu Paralelo* começa após a definição de uma solução inicial viável S_0 . Esta solução inicial é realizada por um algoritmo específico de baixo custo (Seção 3.4.9). A partir desta solução é realizada a busca pela vizinhança orientada pelas estimativas de Taillard, procurando sempre uma melhor solução até que um critério

(Seção 3.4.12) de parada seja satisfeito. Assim, a solução inicial transforma-se na solução corrente **S**.

Através da solução corrente **S** o TS identifica a cada iteração o *caminho crítico* (Seção 3.4.2) e explora um subconjunto **V** de vizinhança $N_{(s)}$ (Seção 3.4.3) deste caminho. Deste subconjunto **V**, apenas um movimento (Seção 3.4.8) é realizado.

Um movimento entre duas operações **a** e **b** (operação subsequente) deve satisfazer as seguintes condições:

- **a** e **b** pertencem ao caminho crítico;
- **a** e **b** não estão inseridas na *lista tabu* (Seção 3.4.6);
- c'_{max} deve ser estimado por **b** através de um procedimento de cálculo (Seção 3.4.11).

O valor de c'_{max} estimado é armazenado em uma lista (Tabela 4) para que as outras *threads* possam ler, identificar e eleger qual é o melhor movimento dentre os movimentos armazenados na lista. A *thread* que possui o menor c'_{max} será a escolhida. Esta informação é do tipo booleano e também é armazenada em “*Movimento Candidato*” na *lista tabu* (Tabela 4).

Com a identificação do *movimento candidato* (eleito), a *thread* que processa a operação eleita realiza um procedimento de movimentação (Seção 3.4.8) com a operação da tarefa antecessora do mesmo robô. Após a conclusão deste movimento, as identificações das duas operações serão inseridas na lista tabu (Tabela 4) impedindo sua movimentação durante o tempo em que estas estiverem inscritas na lista. Após um período ou número de iterações definido como tamanho da lista, estas operações poderão ser novamente movimentadas, de forma a identificar novas soluções.

O procedimento de movimentação, descrito acima, é a geração de uma solução através da busca pela vizinhança $N_{(s)}$ realizando pequenos movimentos locais pertencentes ao caminho crítico. Este processo é a reversão do direcionamento do arco entre as operações **a** e **b** de um robô pertencente ao caminho crítico, reorganizando as operações neste robô: a reversão de um arco (**a**, $RS_{[a]}$) conciste

em reorganizar as operações das máquinas $[RA_{[a]}, a, RS_{[a]}, RS_{[RS[a]]}]$ de um robô (Figura 7) para $[RA_{[a]}, RS_{[a]}, a, RS_{[RS[a]]}]$.

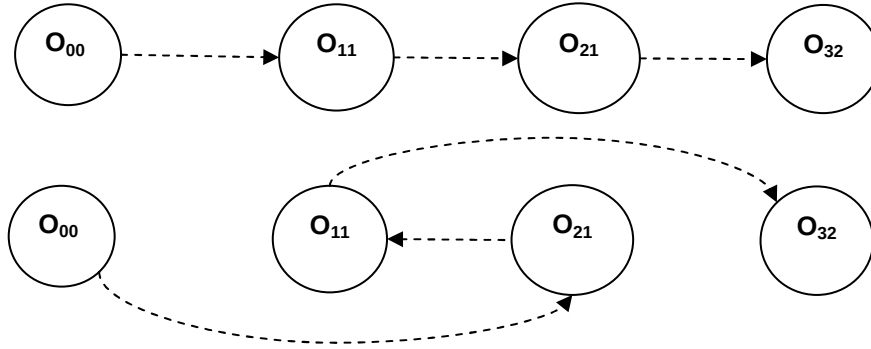


Figura 7: Exemplo de busca pela vizinhança (*Neighborhood*) pela reversão entre arcos de duas operações subsequentes, O_{11} (**a**) e O_{21} (**b**).

Após este movimento entre operações, as demais *threads* pertencentes ao robô são informadas do referido movimento escolhido e estas realizam também o movimento de troca de operações. O próximo passo será para que todas as *threads* identifiquem os seus tempos de entrada para esta nova solução, utilizando o Algoritmo 1.

Com o passar das iterações os melhores movimentos serão inseridos na *lista tabu*, o que vem a ser a estratégia de intensificação de busca pela vizinhança. Neste instante, é possível que todos os movimentos pertencentes ao caminho crítico fiquem proibidos de serem movimentados. Deste modo, um critério de aspiração (Seção 3.4.7) é verificado para contornar o problema de não haver movimentos disponíveis e diversificar para outro espaço de busca: se o c'_{max} for menor que o *makespan* da melhor solução ou S^* , até ali encontrado, este será o eleito. Uma nova solução corrente é gerada ou S' , mesmo que esta seja inferior a solução anterior S , ou seja, quando a busca chega a um ótimo local ela não termina.

Como visto, o procedimento de busca move-se de uma solução para outra fazendo uso de uma estrutura de vizinhança (Seção 3.4.3), procurando a melhor de todas as soluções visitadas ou S^* .

3.4.2 Caminho Crítico

O trabalho de E. Balas (1969) identificou que se forem alterados pares adjacentes entre operações pertencentes ao *caminho crítico* é possível obter como resultado uma outra solução, também viável, possivelmente com melhor *makespan*. Enquanto que se empregado o mesmo método em um *caminho não crítico* e assim resultar em uma solução viável, o *makespan* é maior ou igual à solução anterior.

Conforme descrito na Seção 2.5, quando as direções dos arcos disjuntivos são determinadas, o grafo G passa a ser direcionado e se este grafo for acíclico, esta nova configuração representará uma solução viável para o problema JSP. Caso contrário, um grafo cíclico não estabelece precedência e, portanto o maior caminho é infinito.

A identificação do *caminho crítico* de uma solução S é fundamental para obter-se qualidade na solução em menor tempo de processamento. Este é uma seqüência de operações que identifica um caminho contínuo desde o vértice início do grafo disjuntivo G até o vértice final. O somatório dos tempos de saída de cada operação deste caminho é o *makespan* do problema.

Por exemplo, no escalonamento visto na Figura 4 descrito na Seção 2.6, o *makespan encontrado para aquela solução viável* é de 16 unidades. Identificamos, assim, o caminho crítico como sendo a seqüência de operações O_{10} , O_{20} , O_{21} e O_{22} .

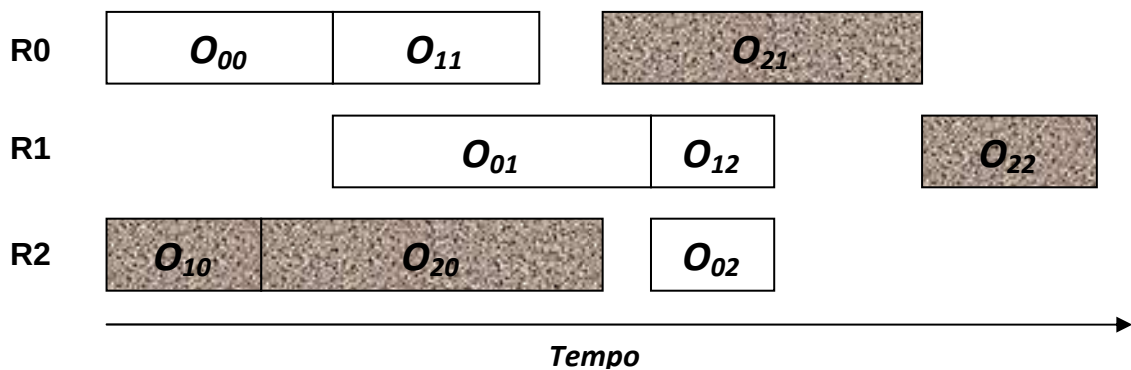


Figura 8: Caminho crítico encontrado em uma solução viável do problema, descrito na Tabela 1, na cor cinza.

No algoritmo paralelo, a *thread* que for identificada pela *thread* vizinha (sucessora) como sendo parte do *caminho crítico* identificará qual das suas duas *threads* vizinhas predecessoras possui tempo final igual ao seu tempo inicial. Assim, o *flag*

da *thread* vizinha que fizer parte do *caminho crítico* é marcado como verdadeiro. Este *flag* é chamado de “Caminho Crítico” na lista tabu (Tabela 4).

A cada iteração um escalonamento da solução é realizado (Seção 3.4.1), sendo assim, o *makespan* desta solução é conhecido e este procedimento de identificação do caminho crítico ocorre a partir da última operação até o início da operação, seguindo a heurística definida no parágrafo anterior.

3.4.3 Estrutura de Vizinhaça para o problema JSP

O processo de *busca local em uma vizinhaça* ou “*neighborhood search*” (**NS**) é um processo iterativo, com o qual é obtida uma nova solução a partir de certos critérios de movimentação de operações candidatas: quando realizada uma troca de operações pertencentes ao *caminho crítico* da solução corrente, é possível alterar todos os relacionamentos de escalonamento desta solução e é possível também que esta troca de operações aperfeiçoe a função objetivo ou *makespan* no referido processo iterativo.

Como descrito anteriormente, a operação de movimentação entre as operações pertencentes ao *caminho crítico* precisa obedecer a certos critérios. A escolha destes critérios é o problema principal para a eficiência do algoritmo e determinam a qualidade da solução. Por outro lado, um processo aleatório de escolha apresenta um custo computacional elevado.

3.4.4 Lista de entrada

Para todas as *threads* há uma identificação ou *id*. Este é utilizado como índice da estrutura de dados que armazena as informações de entrada do problema (JSP). Através da estrutura de dados de entrada é possível identificar as seqüências de tarefas e suas operações, os robôs programados a executarem estas operações e seus respectivos pesos. É possível visualizar esta estrutura através do exemplo a seguir que corresponde aos dados de entrada do grafo **G** descrito no Capítulo 2: um problema com três máquinas ou robôs por três tarefas, totalizando, neste exemplo, nove *threads*:

Tabela 2: Estrutura de dados de entrada do problema JSP com índice da tabela *id* da *thread*.

<i>Id - Thread</i>	Robô	<i>Tarefa</i>	Operação	Peso
0	0	0	0	3
1	1	0	1	5
2	2	0	2	2
3	2	1	0	2
4	0	1	1	3
5	1	1	2	2
6	2	2	0	5
7	0	2	1	6
8	1	2	2	3

A estrutura de dados de *entrada* é estática e é armazenada em *cache*, em uma memória *Constante* do dispositivo GPU, descrito na Seção 4.8 (Hierarquia de Memória). Deste modo é possível otimizar o desempenho do algoritmo, pois este tipo de memória possui baixa latência em relação à memória *Global*.

3.4.5 Lista de saída

A estrutura de dados de *saída*, ou solução, é simples e armazena as informações de tempo de início e tempo final de cada operação. O índice da tabela de solução também é o mesmo empregado pela tabela de entrada ou o atributo *id* da respectiva *thread*.

Ao contrário da estrutura de dados de *entrada*, a estrutura de *saída* ou solução é dinâmica e permite que todas as *threads*, mesmo em diferentes blocos, realizem acessos de leitura e escrita: a estrutura é armazenada na memória *Global* da GPU (DDR3) de modo a garantir que as suas informações sejam visíveis a qualquer *threads* do dispositivo GPU.

Tabela 3: Estrutura de dados de saída ou solução, à esquerda. Neste exemplo é demonstrada uma *solução* viável para o problema JSP da Tabela 2.

Lista de Saída			Lista de Entrada			
Id - Thread	Início da operação	Fim da operação	Robô	Tarefa	Operação	Peso
0	0	3	0	0	0	3
1	3	8	1	0	1	5
2	8	10	2	0	2	2
3	5	7	2	1	0	2
4	11	14	0	1	1	3
5	14	16	1	1	2	2
6	0	5	2	2	0	5
7	5	11	0	2	1	6
8	11	14	1	2	2	3

3.4.6 Lista *tabu*

A palavra “*tabu*” indica qualquer comportamento inaceitável ou proibido e é útil para gerenciar e inibir movimentos na vizinhança da solução corrente **S** recentemente conhecidos, na tentativa de se buscar novas soluções, evitando-se, assim, os ciclos existentes no grafo ou a repetição da busca no mesmo ponto. Este comportamento é representado por uma *lista tabu* com $|L|$ movimentos proibidos.

A *lista tabu* contém os movimentos reversos aos últimos $|L|$ movimentos aceitos e funciona como se fosse uma fila FIFO (“*first-in first-out*”): quando um novo movimento é aceito à lista, o movimento mais antigo é excluído dela.

O índice da *lista tabu* é o mesmo utilizado para as listas de *entrada* e de *saída*. Quando uma operação é movimentada, a informação de “*tempo tabu*” é incrementada até o limite imposto por $|L|$ movimentos aceitos. Este tempo, diferente de zero, impede os movimentos de operações no instante da seleção do movimento candidato. O tamanho da *lista tabu* $|L|$ é inicializado com $(t + r)/2$, onde t é o número de tarefas e r o número de robôs, por apresentar bons resultados (TAILLARD, 1994).

O tamanho da *lista tabu* é flexível ou dinâmico em função dos resultados de busca local:

- Diminui-se o *tempo tabu* | **L** | quando se encontra uma solução melhor que a anterior, intensificando a busca na região;
- Aumenta-se o *tempo tabu* | **L** | ao encontrar soluções piores que a solução anterior, diversificando a busca local, para procurar em outras regiões.
- Se a solução é a melhor solução encontrada, o tamanho da lista é dimensionado para um.

Um valor mínimo e máximo é uniformemente escolhido para o tamanho máximo e mínimo da *lista tabu*, procurando evitar a ciclagem entre operações (TAILLARD, 1994):

$$L_{min} = [0,8 L] \text{ e } L_{max} = [1,2 L].$$

É descrito na Seção 3.4.11 que as *threads* estimam o custo para a sua solução vizinha e inserem o valor estimado c'_{max} na *lista tabu* (Tabela 4), indicando, assim, seu candidato. O *movimento candidato* é eleito quando a operação fizer parte do *caminho crítico*, este não é *tabu* (a exceção do critério de aspiração) e a solução estimada c'_{max} (solução de cada *thread*) seja a melhor delas.

Tabela 4: Estrutura de dados da *lista tabu*. Neste exemplo é demonstrado o estado de uma *solução* viável para o problema JSP descrito na Tabela 2.

Lista Tabu					Lista de Entrada			
Id - Thread	Caminho Crítico	Movimento Candidato	Tempo Tabu	c'_{max}	Robô	Tarefa	Operação	Peso
0	0	0	0	INFINITO	0	0	0	3
1	0	0	0	INFINITO	1	0	1	5
2	0	0	0	24	2	0	2	2
3	0	0	2	16	2	1	0	2
4	1	0→1	0→1	19	0	1	1	3
5	1	0	0	26	1	1	2	2
6	1	0	2	INFINITO	2	2	0	5
7	1	0	0→1	24	0	2	1	6
8	1	0	0	24	1	2	2	3

Repare que a *thread* ao indicar o seu candidato poderá torná-lo, alguns deles, inelegível ao informar o c'_{max} como INFINITO, por exemplo, quando a operação é escalonada no início do robô e não é possível realizar o movimento de troca entre operações.

Quando o movimento candidato é aceito, obedecendo as regras acima, o status do “*Movimento Candidato*” é alterado para *verdadeiro*. No exemplo anterior (mostrado na Tabela 4) , a $thread_4$ e a $thread_7$ irão posteriormente proceder ao movimento entre as operações O_{21} e O_{11} (Figura 9 e 10), movendo a solução corrente para uma solução vizinha que foi escolhida entre todas as soluções. Como visto, o *tempo tabu* é incrementado de zero para um, inserindo estas duas operações na *lista tabu* durante o tempo de $|L|$ iterações.

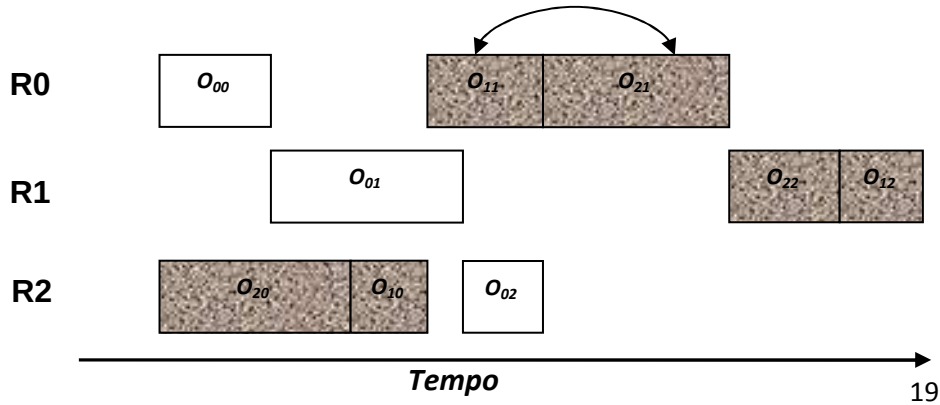


Figura 9: Exemplo de perturbação no caminho do grafo G através da troca da operação O_{11} pela sua operação precedente O_{21} pertencentes ao caminho crítico.

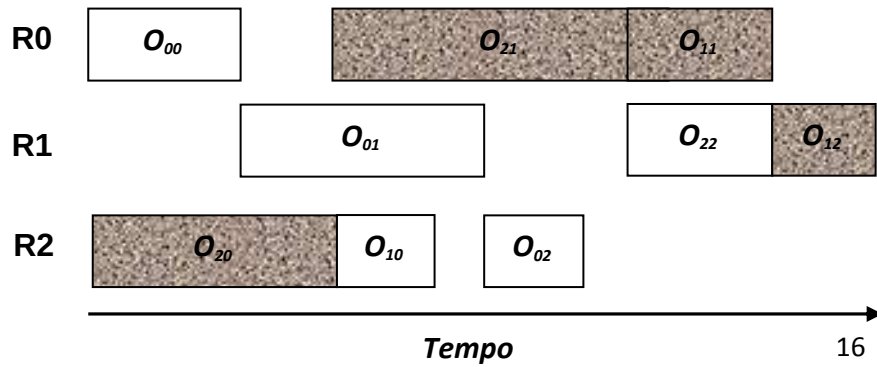


Figura 10: Exemplo de resultado entre o movimento entre as operações O_{11} e O_{21} pertencentes ao caminho crítico.

3.4.7 Critério de Aspiração

Durante o processo de busca na vizinhança, pode ocorrer uma condição em que todos os movimentos candidatos serão *tabu*, impedindo o próximo movimento e, deste modo, interrompendo a busca. Este problema é restritivo, impede o retorno a uma solução já gerada e também a outras soluções ainda não geradas. Uma solução para permitir que a busca continue é contornar este problema com a adoção

de outros critérios para cancelar a condição de *tabu* do melhor movimento. O critério adotado, chamado de *Critério de Aspiração por Objetivo*, consiste em permitir que ocorram trocas que geram um valor de *makespan* menor que os encontrados até então, mesmo que este movimento esteja em *tabu*.

3.4.8 Movimentação entre operações

No exemplo anterior, a *thread*₇ que processa a operação O_{21} do robô R_0 se identifica como sendo o movimento candidato eleito e, deste modo, realiza um movimento de troca com a operação precedente O_{11} . Esta é processada pela *thread*₄ e ambas as operações são incluídas em “*tempo tabu*” na *lista tabu* e impedidas de se movimentarem até a sua saída da lista ou se o critério de aspiração for atingido.

Após a troca das operações vizinhas entre as operações O_{11} e O_{21} as *threads* reajustam seus tempos de entrada e de saída (inicial e final), atualizando a *lista de saída* (Tabela 3), e obtendo o escalonamento de uma nova solução S a qual possui seu próprio caminho crítico. Podemos ver o resultado deste exemplo através da Figura 10 onde uma melhor solução (*makespan* 16) foi encontrada em relação à solução anterior (*makespan* 19).

3.4.9 Solução inicial

O primeiro procedimento utilizado no algoritmo paralelo é encontrar uma solução inicial viável. A heurística desenvolvida utiliza o escalonamento semi ativo o qual é caracterizado pelo fato de que o tempo de início de qualquer operação de qualquer tarefa sobre qualquer robô é determinado pela conclusão da operação de uma tarefa precedente no mesmo robô ou a conclusão da operação antecessora na mesma tarefa (SHOUICHI e YAMADA, 2007). A exceção a ser considerada é para a operação inicial que não possui operação precedente.

Ao gerar a solução inicial, as primeiras operações de cada tarefa são as candidatas imediatas para o escalonamento e o critério de despacho adotado é a ordem crescente dos tempos de entrada para as operações.

A heurística procura uma solução utilizando os recursos da computação paralela: cada *thread* procura resolver o escalonamento em cada operação existente no

problema (JSP), conforme descrito na Seção 3.4.1, ou seja, para encontrar os tempos de escalonamento de cada operação, cada *thread* irá realizar um procedimento de cálculo para o *tempo inicial* (entrada) de modo a gerar uma solução viável.

$$t_{i[i]} = \max(t_{TA[i]} + p_{TA[i]}, t_{RA[i]} + p_{RA[i]})$$

Algoritmo 2 - Solução Inicial viável

```

Inicializa tempos de entrada das operações com 0 para thread i;
Inicializa tempos de saída das operações com INFINITO para thread i;
para cada thread i = 0, ..., (t x r - 1) faça
    inicializa n para o espaço de memória cache para thread i;
    para j = 0, ..., (t x r - 1) faça
        se Robô[j] é o mesmo Robô da thread i
            R[n].id = j;
            R[n].t = se o robô possui operação zero então seu tempo de entrada
                será zero senão será o tempo final da operação
                predecessora, se existir;
            n++;
    fim se
    fim para
    ordena a lista pelo tempo inicial (entrada) R[n].t, ascendente;
    calculaTempoEntrada da thread i;
fim para;
```

A cada iteração *i* é feito uma ordenação dos tempos de entrada da seqüência de robôs, utilizando o método de ordenação por bolha ou “*bubble-sort*”. As informações dos tempos inicial (entrada) e final das operações escalonadas para cada Robô são lidas da *memória Global* do dispositivo (lista de saída – Tabela 3), de alta latência, transferidas e armazenadas em *R[n]*, onde *n* é um espaço de memória *cache* do dispositivo (memórias compartilhadas) de baixa latência alocadas para cada *thread*. Estas estarão prontas para a próxima fase do algoritmo que é a identificação do *caminho crítico* e a identificação do movimento candidato entre as *threads*.

3.4.10 Solução vizinha

Após a definição da solução inicial, o algoritmo *tabu* irá realizar sucessivas buscas na vizinhança da solução corrente *S* realizando operações de troca entre as operações eleitas, pertencentes ao caminho crítico. Para cada busca, é feito um restabelecimento dos tempos de entrada para cada operação, processada por uma *thread*, definindo uma nova solução *S*.

Semelhante ao algoritmo que encontra a solução inicial, este processo de troca de informações entre todas as *threads* ocorre durante um ciclo de $t \times r$; onde, como informado anteriormente, t é o número de tarefas e r o número de robôs.

Algoritmo 3 – Solução Vizinha

```

inicializa  $n$  para o espaço de memória cache para thread  $i$ ;
para  $j = n, \dots, (n + r - 1)$  faça
     $R[j].t$  = tempo inicial da operação armazenada na memória global;
fim para
para  $k = 0, \dots, (t \times r - 1)$  faça
    calculaTempoEntrada da thread  $i$ ;
fim para;
```

Como visto no artigo de Taillard (1994) é possível realizar uma estimativa para o *makespan* das soluções vizinhas de S aplicando o cálculo semelhante ao definido para o tempo de entrada da operação (Seção 3.4.1), para todas as operações, em paralelo.

3.4.11 Solução candidata

Utilizando a estratégia de Taillard o algoritmo indica de forma rápida e fácil uma estimativa de custo para cada solução vizinha S' da solução corrente S , lembrando que o escalonamento é do tipo semi ativo. Cada *thread* realiza o procedimento de definição para a indicação da sua solução candidata.

Sejam duas operações a e b como operações adjacentes (vizinhas) e pertencentes ao caminho crítico. A operação b é sucessora da operação a ou $b = RS_{[a]}$. Assim, cada *thread* estima sua própria solução e em conjunto com todas as informações das outras *threads* poderá ser eleito qual é o melhor movimento a ser executado para encontrar uma nova solução. Os valores estimados por cada *thread* poderão ser obtidos por um simples cálculo e o seu resultado é armazenado na *lista tabu* (Tabela 4) em c'_{max} :

1 - Para cada operação processada pela *thread* i ($1 \leq i \leq N$).

- t'_b – novo tempo de entrada da operação b (“*release time*”) após a troca de b por a .
- t_{RAa} – tempo final da operação processada (se existir) no robô antecessor de a ;
- $t_{TA b}$ – tempo final da operação (se existir) pertencente a operação antecessora de b ;

- t'_a – novo tempo de entrada da operação **a** (“*release time*”) após a troca de **b** por **a**.
- p_b - Tempo (peso) de processamento de **b**;
- t_{TAa} –tempo final da operação (se existir) pertencente a operação antecessora de **a**;
- $c'a$ – novo comprimento para o caminho de **a** até $N + 1$ (“*length of de tail*”).
- C_{RSb} – caminho de **b** até $N + 1$ para a operação processada no robô sucessor de **b**.
- C_{TSa} – caminho de **a** até $N + 1$ para a operação (se existir) pertencente a operação sucessora de **a**;
- p_a - Tempo (peso) de processamento de **a**;
- $c'b$ – novo comprimento para o caminho de **b** até $N + 1$ (“*length of de tail*”).
- C_{TSb} – caminho de **b** até $N + 1$ para a operação (se existir) pertencente a operação sucessora de **b**;

2 - Para cada thread i haverá a seguinte identidade:

- $t'_{i[b]} = \max(t_{RA[a]}, t_{TA[b]})$
- $t'_{i[a]} = \max(t'_{i[b]}, t_{TA[b]})$
- $c'_{i[a]} = \max(c_{RS[b]}, c_{TS[a]} + p_a)$
- $c'_{i[b]} = \max(c'_{i[a]}, c_{TS[b]})$

O valor para o *makespan* estimado e candidato que poderá ser elegível é:

$$c'_{max} = \max(t'_{i[b]} + c'_{i[b]}, t'_{i[a]} + c'_{i[a]})$$

Este valor encontrado é um novo valor estimado para o *makespan* que representa o movimento entre as operações **a** e **b** e deste modo é possível identificar e dirigir a busca na estrutura de vizinhança para encontrar a melhor solução.

3.4.12 Critério de Parada

O Simulador utiliza como critério de parada um número fixo de iterações definido pelo pesquisador e assim este poderá acompanhar graficamente a evolução do algoritmo em todo o processo de busca local. O pesquisador poderá verificar a eficiência do método estudado, assim como, poderá parar a simulação em um ponto da busca, delimitando apenas o número de iterações através da interface web do Simulador.

3.4.13 Invocação do *Kernel* pelo CUDA

Como será visto na Seção 4.11.1 o procedimento de chamada ao *kernel* se dá através do computador hospedeiro ou *Host*. Ao invocar o *kernel*, duas informações deverão ser passadas ao dispositivo GPU, além das listas inicializadas *tabu*, *entrada* e *saída*: são os parâmetros do problema JSP como a informação da dimensão da *grid* e do número de *threads* por bloco. O número de *threads* por bloco é definido pelo tamanho do problema (JSP) e pelo número de blocos definido pelo usuário. Por exemplo, um problema ou instância com o tamanho de entrada de seis tarefas por seis robôs compõem, por esta definição, um número de trinta e seis *threads* que serão necessárias (*numThreads*) para processar o problema JSP. Se o usuário definir que estas trinta e seis *threads* serão distribuídas em apenas um bloco (*nbloco*), o algoritmo calculará o número de *threads* por bloco assim como o tamanho da *grid*. No Simulador, este é um parâmetro de ambiente para que o pesquisador possa estudar e observar o comportamento do método em evidência, através da manipulação desta informação.

Com esta configuração é possível alocar mais recursos computacionais com a disponibilização de mais blocos ou mais processadores, de acordo com o tamanho de entrada do problema JSP.

Algoritmo 4 – (Invocação do *kernel*)

inicializa as listas *tabu*, de entrada e saída;
transfere as listas do host para o dispositivo;
buscaTabu<<<nbloco, blockSize>>>(**S^*** , **S** , **L** , **max** , *númeroRobôs*, *númeroTarefas*);
transfere S^* , a lista de saída com a melhor solução do dispositivo para o host;

```

__global__ void
buscaTabu( $S^*$ ,  $S$ ,  $L$ ,  $max$ , númeroRobôs, númeroTarefas) {
     $i \leftarrow 0$ ;                               /* inicializa a evolução mais recente que forneceu  $S^*$  */
     $L \leftarrow 0$ ;                             /* inicializa a lista tabu */
    Inicializa  $f_{(S)}$  com INFINITO;
    Inicializa  $S$  com a solução viável, inicial;
    Inicializa  $f_{(S)}$  obtida pela solução inicial viável;
    Inicializa a função de aspiração  $A$ ;
    para  $e = 0, \dots, (e < max)$  e  $(id < númeroRobôs * númeroTarefas)$  faça
        se  $f_{(S)} < f_{(S^*)}$  então
             $S^* \leftarrow S$ ;
             $i \leftarrow e$ ;
            Atualiza a função de aspiração  $A$   $f_{(S^*)}$ ;
        fim-se;
        identifica o caminho crítico de  $S$ ;
        encontra a solução candidata de  $V \subseteq N_{(S)}$  indicada pela thread;
         $S' \leftarrow S \oplus k$  o melhor candidato de  $V \subseteq N_{(S)}$ 
            tal que o movimento  $k$  pertença ao caminho crítico
            e que o movimento  $k$  não seja tabu ( $k \notin L$ )
            ou  $S'$  atenda a condição de aspiração ( $f_{(S')} < A f_{(S^*)}$ );
        Atualiza a Lista tabu  $L$ ;
        Atualiza o tamanho da Lista Tabu  $L$ ;
    fim para
    retorne  $S^*$ ;
}

```

Capítulo 4

Computação Paralela - CUDA

4.1 Introdução

Este capítulo tem por objetivo a apresentação de alguns conceitos básicos da computação paralela, e encontra-se dividido em duas partes. A primeira aborda inicialmente a apresentação da arquitetura de dispositivos de computação paralela, especificamente disponíveis em Unidade de Processamento Gráfico de Propósito Geral, “*General Purpose Graphics Processing Unit*” (GPGPU) ou GPU para simplificar.

A segunda parte, seguindo a mesma ordem de apresentação da arquitetura da GPU, aborda o método de processamento de dados utilizando extensão da linguagem C/C++ em uma GPU, em particular a arquitetura CUDA da NVIDIA.

A computação paralela é uma técnica conhecida que faz uso de várias Unidades de Processamento (UP), objetivando executar uma computação mais rápida. Seu princípio básico está no processo organizado em resolver um problema: este problema pode ser dividido em problemas menores e os mesmos serem resolvidos simultaneamente através de algum tipo de gerenciamento.

Uma computação paralela pode ser dividida em tarefas onde cada tarefa será realizada por uma *thread* individual. A expressão *thread*, utilizada neste contexto, refere-se à generalização de uma instância de execução de um programa particular, utilizando uma entrada de dados definida (BUCK e BASTIAAN, 2008).

Threads paralelas são executadas simultaneamente, utilizando diferentes processadores paralelos ou núcleos, permitindo que mais trabalhos de processamento sejam completados em uma determinada quantidade de tempo.

Existem diversas arquiteturas de processadores que suportam o processamento paralelo. No início, cada arquitetura utilizava múltiplos processadores discretos e uma rede de interconexão (AUDE *et al.*, 1996).

Arquiteturas modulares paralelas foram desenvolvidas na UFRJ, em meados da década de 90. A arquitetura MIMD foi utilizada no projeto Multiplus e esta suportava

até 1024 processadores SPARC. O Multiplus era composto por até 128 *clusters* de até oito elementos de processamento cada, através de uma rede de interconexão, compartilhando uma memória global distribuída. Esta arquitetura facilitava a criação de *threads* com uso de primitivas desenvolvidas para a sincronização, integradas a um sistema operacional “*Unix-like*”, escrito em C, especializado em operações multitarefas, o Multiplex (AUDE *et al.*, 1996).

4.2 Motivação

- “GPUs chegaram a um ponto onde as aplicações do mundo real serão facilmente implementáveis e executarão muito mais rápido que os sistemas multi-cores”.
- “O futuro da arquitetura será um sistema híbrido com GPUs many-cores trabalhando conjuntamente com CPUs multi-cores”. Jack Dongarra - Professor, Universidade de Tennessee; Autor de “*linpack*”.

Os recursos para aumentar o desempenho do *hardware* em um *chip* estão no seu limite, e estes foram abordados por ZHIRNOV *e tal*, ao publicarem o artigo “Limits to Binary Logic Switch Scaling--A Gedanken Model”. Ao constatar que o constante aumento da frequência de operação e da redução sistemática da espessura da camada de silício obrigaria a indústria a buscar tecnologias alternativas para a fabricação de microprocessadores (ZHIRNOV *e tal*, 2003). Previram ainda que para diminuir o tempo de resposta de um algoritmo será necessária a execução deste em paralelo, reduzindo a necessidade de utilizar componentes de alta velocidade.

Como visto no capítulo 1, o problema *Job Shop* apresenta a tendência de crescimento exponencial do tempo de computação. Com este fato, para diminuir o referido tempo computacional, identifica-se uma nova abordagem de solução ao referido problema, utilizando Processador Gráfico (GPU) de baixo custo. Este possui muitos núcleos de processamento paralelo e apresenta um modelo de computação SIMD, o qual veremos na próxima Seção, perfeitamente adequado ao presente estudo, que têm por objetivo simular algoritmos em solução de escalonamento de tarefas “*Job Shop*” em um ambiente *web* utilizando arquitetura paralela.

4.3 Taxonomia de Flynn

Recentemente, múltiplos núcleos de processamento têm sido fabricados no mesmo *chip*. Estes núcleos podem ser controlados de várias formas. Uma classificação de longa longevidade para estes modelos de arquitetura de computadores foi proposta por Michael J. Flynn em 1996. Esta classificação, que ficou conhecida como “*Taxonomia de Flynn*”, define ao todo quatro classificações que são caracterizadas pelo número de instruções concorrentes (ou controle) e pelo conjunto de dados:

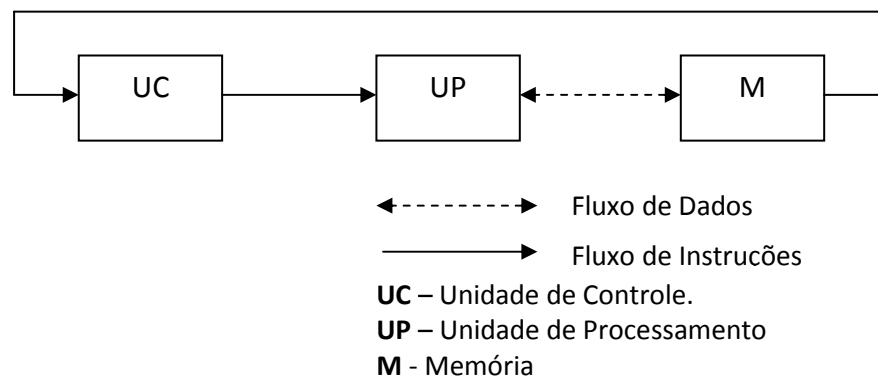


Figura 11: Modelo de arquitetura SISD.

- SISD (“Single Instruction Single Data”) – Esta categoria define a primeira arquitetura de computadores, e é das mais importantes: na maioria dos computadores pessoais já desenhados e construídos até recentemente, nos diversos dispositivos embarcados, existentes em quase todos os equipamentos eletrônicos. A *única instrução* refere-se ao fato de existir apenas uma instrução sendo executada na CPU, durante certo ciclo de relógio. *Dado único* significa, analogamente, que um - e somente um - fluxo de dados estará sendo utilizado como entrada durante certo ciclo de relógio. Estes fatores tratam de duas importantes características da categoria SISD: Serial – as instruções podem ser executadas uma após a outra e este tipo de execução serial é comumente chamado de processamento em série, em oposição ao processamento paralelo onde as várias instruções podem ser tratadas em paralelo; Determinística – como cada instrução é executada individualmente pode-se recriar exatamente o passo a passo do processo posterior, sem alterar o resultado final.

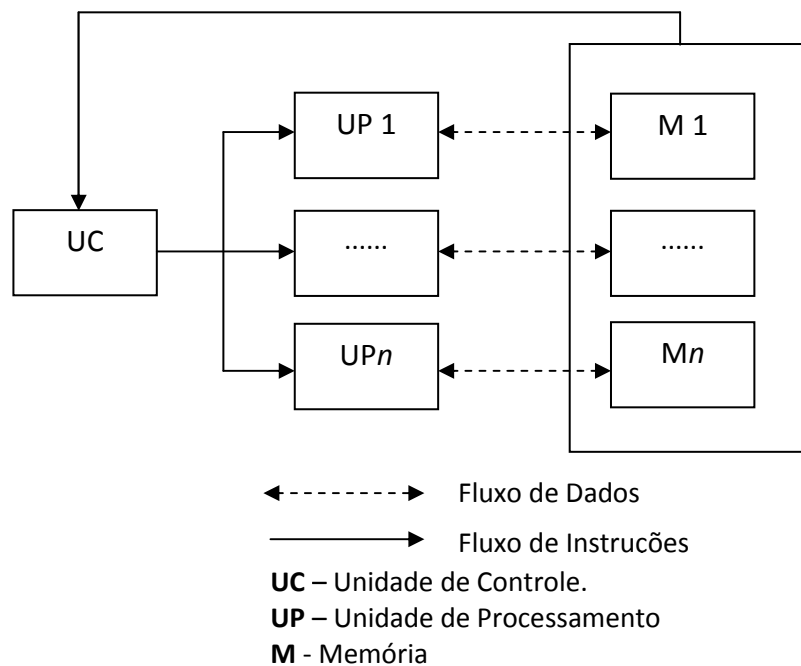


Figura 12: Modelo de arquitetura SIMD.

- SIMD ("Single Instruction Multiple Data") – Esta é uma categoria importante, e sua aplicação encontra-se disponibilizada nas arquiteturas das GPUs. Define a classe de computadores que é capaz de aplicar o mesmo fluxo de instrução a um múltiplo fluxo de dados simultaneamente. Para certas classes de problemas, por exemplo, aqueles conhecidos como problemas paralelos, este tipo de arquitetura é perfeitamente adequado, e obtém com isto altas taxas de processamento. Isto se deve ao fato que os dados podem ser divididos em várias partes individuais, e os múltiplos processadores (todos idênticos) executarão a mesma instrução sobre estes dados, simultaneamente. Características da categoria SIMD: Síncrona – Este tipo de sistema é geralmente considerado como síncrono o que significa que são construídos de forma a garantir que todas as unidades recebam a mesma instrução ao mesmo tempo, e assim todos serão potencialmente capazes de executar a mesma operação simultaneamente; Determinística – Estas arquiteturas são determinísticas porque há apenas uma instrução sendo executada em determinado instante no tempo, e o resultado é garantido exatamente em todas as etapas do algoritmo; adaptável ao nível de paralelismo em

um fluxo único de instruções onde este é executado por várias unidades iguais de processadores em diferentes partes de dados, todos do mesmo tipo e ao mesmo tempo, alcançando assim o paralelismo.

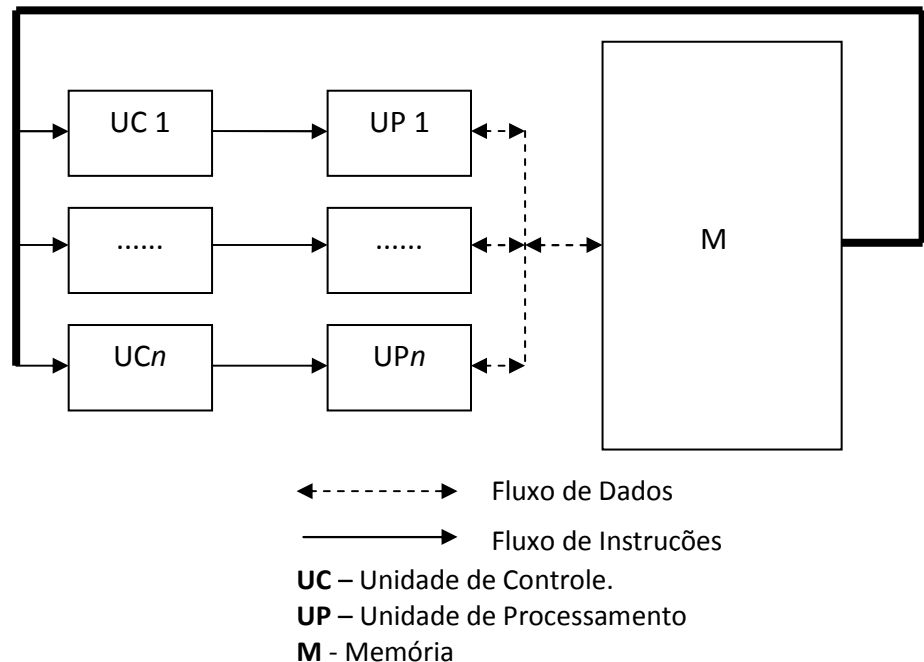


Figura 13: Modelo de arquitetura MISD.

- MISD (“Multiple Instruction Single Data”) Poucos exemplos concretos de computadores podem ser aplicados nesta categoria. Alguns problemas poderiam ser encaixados nesta categoria como sendo máquinas de propósitos específicos, como por exemplo, filtros que operam em múltiplas frequências em um fluxo de dados constante ou vários algoritmos criptográficos tentando decifrar uma única mensagem codificada. Ambos são exemplos deste tipo de categoria de processamento, onde múltiplas instruções independentes são aplicadas ao mesmo fluxo de dados, simultaneamente.

- MIMD (“Multiple Instruction Multiple Data”) Esta categoria poderá ser a próxima evolução das arquiteturas a serem utilizadas nas gerações de computadores onde o paralelismo será obtido a partir de vários fluxos de instruções processando vários fluxos de dados ao mesmo tempo. Esta é a categoria mais genérica, e que pode representar

qualquer uma das outras três categorias. Síncrona ou assíncrona - Esta arquitetura pode executar instruções síncronas ou assíncronas, estritamente executam o mesmo conjunto de instruções paralelas ao mesmo tempo, ou executam estas instruções em tempos diferentes, dependendo da estratégia do algoritmo utilizado. Determinístico ou não determinístico – Sistemas MIMD são potencialmente capazes de possuir o comportamento determinístico, que é exatamente a reprodução dos passos do algoritmo para as mesmas entradas de dados. Mas se for importante que o sistema se comporte não deterministicamente, esta categoria de arquitetura pode desempenhar o processamento paralelo.

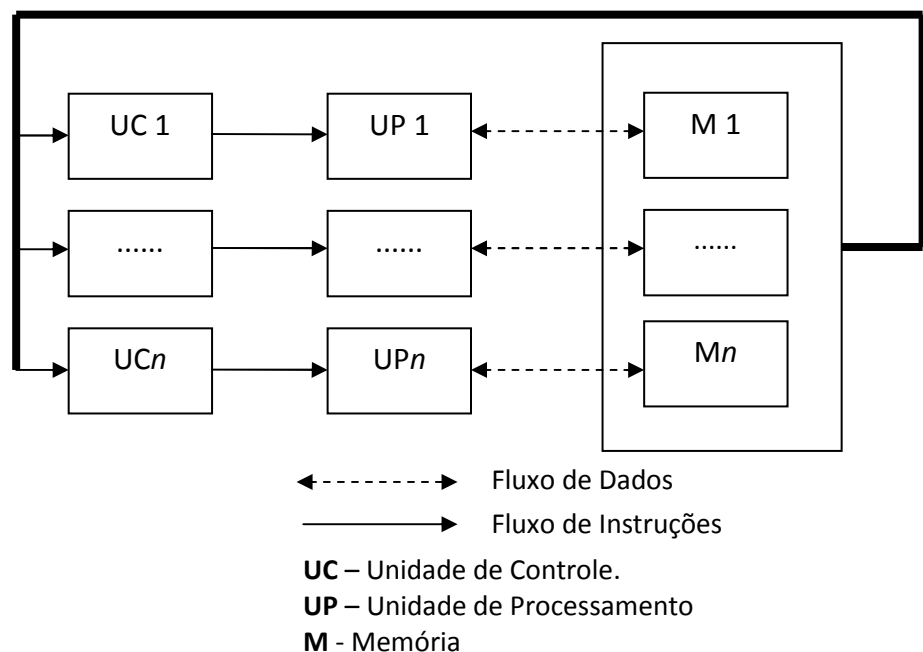


Figura 14: Modelo de arquitetura MIMD.

4.4 Gerações anteriores de programação para GPU

Gerações anteriores de GPUs consistiam em “*pipe lines*” altamente especializados, para os quais qualquer aplicação não-gráfica deveria ser mapeada através de interfaces gráficas (APIs) como OpenGL e Direct3D, impondo uma alta curva de aprendizagem para os desenvolvedores iniciantes e o alto custo para aplicações não gráficas (HUANG *et al.*, 2008).

Processadores gráficos GPUs utilizam arquiteturas SIMD (HUANG *et al.*, 2008). Esta arquitetura permite uma alta taxa de transferência de dados para aplicações de renderização de imagens.

Nesta arquitetura gráfica SIMD, as *threads* são utilizadas em uma estrutura de duas ou três dimensões que compõe uma matriz ou *grid* e executarão, como por exemplo, algoritmos do tipo “*Vertex Shader*” e algoritmo “*Pixel Shader*”⁵.

Geralmente cada aplicação requer a execução de um mesmo programa e um grande número de objetos (vértices ou primitivas) independentes (NICKOLLS *et al.*, 2008). Assim, cada objeto é processado independentemente de todos os outros, utilizando a mesma seqüência de operação. Isto disponibiliza aos desenvolvedores uma arquitetura de *hardware* complexa.

A arquitetura SIMD requer o processamento de *threads* em paralelo, e estas devem ser preferencialmente independentes uma das outras. Por exemplo, deve ser evitado que uma *thread* utilize os dados gerados por outra *thread* que esteja executando paralelamente (NICKOLLS *et al.*, 2008) a não ser que haja um mecanismo de coordenação para isso, o que faz com que o programa fique mais complexo. Este cuidado tem como objetivo não gerar dependência de dados, e assim manter a consistência e eficiência de execução destes.

As abstrações desenvolvidas com o paralelismo nas GPUs fornecem uma granulagem-fina de paralelismo de dados, onde as *threads* trabalham cooperativamente processando e mapeando estes dados. O paralelismo de tarefas com independência e o paralelismo de *threads* aninhadas fornecem uma granulagem grossa para o paralelismo. São meios fundamentais para a execução da mesma instrução nos diversos processadores ou núcleos existentes na GPU.

Em outros casos, o compartilhamento de dados é disponibilizado internamente na GPU: para algumas instâncias da SIMD, é disponibilizado acesso a estas memórias *cache* internas ou registradores globais⁶ que podem ser acessados por todos os núcleos de processamento.

⁵ - “*Vertex e Pixel Shader*” são técnicas utilizadas em computação gráfica para produzir texturas através de polígonos independentes, de forma a modelar um objeto graficamente.

⁶ Algumas máquinas paralelas MIMD passam mensagens entre processadores, utilizando a rede de interconexão, ou memória compartilhada. Por outro lado, um núcleo da GPU pode registrar um dado em uma memória *cache* que é subseqüentemente lido por outro núcleo de processamento.

Tradicionalmente, os ambientes de programação para GPUs possuem soluções específicas (BUCK e BASTIAAN, 2008), cujo objetivo final é a geração de imagens.

Linguagens semelhantes à CG⁷, anterior ao CUDA (a ser apresentado na Seção 4.11), e “*High Level Shader Language*” (HLSL) desenvolvida pela Microsoft Corporation, permitem que os usuários utilizem “*Vertex*” e “*Pixels*” de forma compartilhada, em um ambiente que é similar ao ambiente de programação C/C++.

Estas soluções de trabalho são boas para aplicações gráficas específicas, como por exemplo, videogames, mas não são tão boas para a computação em propósitos gerais.

O desafio no uso da computação paralela está no desenvolvimento de aplicações de *software* que utilizem todo o paralelismo existente no crescente número de núcleos dos processadores atuais, seguindo o mesmo passo que as aplicações gráficas 3D vêm realizando com as GPUs “many-cores”, com grande variedade de número de núcleos.

O rápido crescimento na indústria de vídeo games exerceu e ainda exerce uma forte pressão econômica no mercado, o que força as constantes inovações na arquitetura dos computadores, fazendo GPUs mais flexíveis e com maior poder de processamento, como:

- Podem ser programáveis através de uma linguagem de alto nível;
- Suportam 32-bits e 64-bits de precisão em ponto flutuante IEEE-754;
- Possuem Unidade Lógica (add, sub, mul e mad);
- Possuem unidade de comparação e desvio;
- GPUs estão na maioria dos PCs e estações de trabalho.

4.5 GPUs *many core*

Como visto na Seção anterior, antes da existência das GPUs, havia diferentes aceleradores gráficos 3D, que eram relativamente limitados no seu processamento e dependiam do processador central – host – para as funções de iluminação e transformação de vértices - “*transform and lighting*” ou T&L.

⁷ C for Graphics é uma linguagem de programação da NVIDIA derivada do ANSI C para suporte ao design gráfico, aproveitando os recursos do hardware NVIDIA.

Atualmente, as três maiores empresas⁸ que fabricam GPUs são: a ATI Technologies, recentemente adquirida pela AMD, NVIDIA e a Intel (GPU integrada nos chipsets).

Dirigidas por uma demanda de mercado sempre crescente para o processamento em tempo real e alta definição de gráficos 3D, principalmente para jogos de videogames, as GPUs programáveis são um tipo de microprocessadores especializados em processar gráficos para computadores pessoais e videogames. Ou seja, as unidades de processamento gráfico utilizam em grande escala a manipulação vetorial e operações de cálculo de ponto flutuante.

A eficiência alcançada pela GPU está no uso da arquitetura de processamento paralelo de alto desempenho, através de um alto paralelismo, uso de múltiplas “*threads*” e vários processadores “*many-core*”.

Outra eficiência apresentada vem através da controladora de memória da GPU que foi projetada para uma alta taxa de transferência - “*throughput*” - entre a memória “*on-board*” do dispositivo gráfico e a GPU: as GPUs possuem baixa frequência de relógio (~600 MHz) em relação à memória da CPU (~1,8 GHz), mas apresentam uma latência reduzida para acessar os dados sequencialmente, permitindo o “*prefetch*” e o “*pipe line*”. Ao contrário, a CPU possui uma alta frequência de relógio (~3 GHz) e a velocidade de acesso à memória é bem menor (~533 MHz), conseqüentemente sofrerá “*stalls*” porque a taxa de transferência de memória para a CPU é inadequada.

A principal razão por trás desta evolução é que a GPU é uma unidade especializada na geração de gráficos renderizados. Por esta razão foram projetadas com mais transistores devotados (como pode ser visto na Figura 15) ao processamento de dados do que os reservados para armazenamento de memória *cache* e unidades para controle de fluxo, largamente utilizados nas CPUs (NVIDIA, 2008).

Mais especificamente, a GPU é especialmente adequada para alguns problemas que manipulam amplamente o endereçamento de um array para cálculo aritmético, por exemplo, e pode ser expressa em processamento de dados paralelos: a mesma instrução de um programa é executada em muitos dispositivos ou núcleos de dados em paralelo, internamente na GPU, com alta intensidade no uso do cálculo vetorial

⁸ As três empresas utilizam a TSMC - “Taiwan Semiconductor Manufacturing Company, Limited” para o processo de difusão e industrialização de seus produtos.

ou matricial. Por este motivo, no qual a mesma instrução de um programa é executada para cada elemento de dado, há pouca necessidade de haver um controle de fluxo sofisticado; e porque é executada em muitos elementos de dados e com alta intensidade de cálculo aritmético, a latência ao acesso de dados pode ser escondida com cálculos instanciados em uma cache de dados.

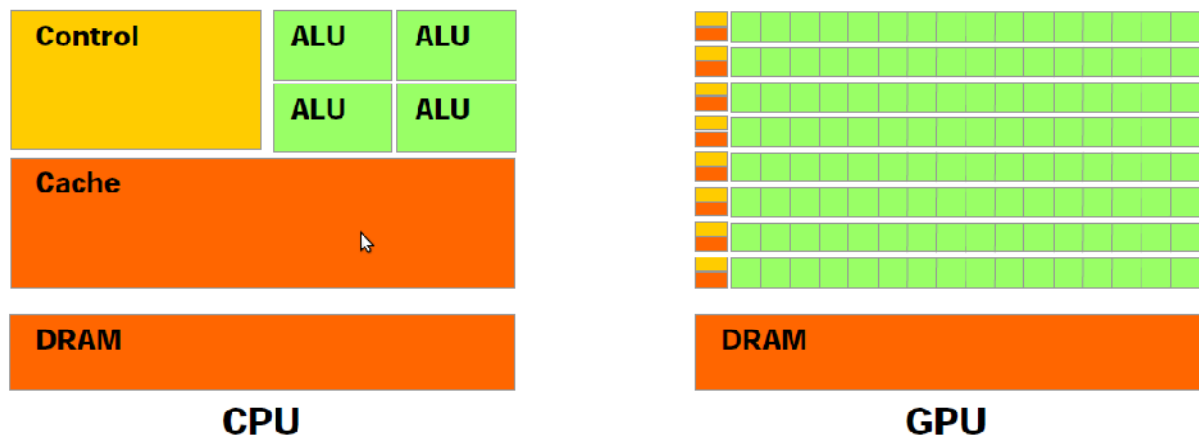


Figura 15: Diferenças entre a distribuição dos componentes em uma CPU e em uma GPU.

Até agora, entretanto, o acesso a todos os recursos computacionais existentes na GPU e a sua eficiência estavam sendo voltados para as aplicações gráficas. Mas a partir de junho de 2007 a NVIDIA expandiu a sua arquitetura de *software* para as aplicações não gráficas disponibilizando o guia de programação - CUDA, v 1.0, para as GPUs modelos G80 (NVIDIA, 2008).

4.6 Arquitetura em *threads*

O processamento de dados paralelos mapeia os elementos de dados em um conjunto de instruções chamadas *thread*, antes do processamento paralelo. Muitas das aplicações que processam grandes conjuntos de informações, como arrays de dados, podem utilizar este modelo de programação de dados paralelos para aumentar a velocidade de computação.

Este trabalho utilizou a unidade processadora gráfica – GPU – da NVIDIA, modelo Geforce 9800 GT, como o *hardware* responsável para a execução do algoritmo para escalonamento de tarefas utilizado no Simulador - “*Job Shop*” ou JSP. Por uma questão de razoabilidade no custo *versus* benefício: baixo custo ($\approx$ \$2,00 por núcleo); alto desempenho; escalabilidade; portabilidade e facilidade para programar.

Este modelo possui 112 núcleos, o que é uma quantidade razoável de núcleos⁹ para ser utilizado nos experimentos. Cada um destes núcleos existentes na GPU são processadores independentes e realizam tarefas cooperativamente em paralelo.

O modelo de programação desenvolvido pela NVIDIA, para aplicações não-gráficas, utiliza a linguagem ANSI C de programação de propósito geral. Esta linguagem é difundida na indústria de *software* por ser estruturada, imperativa, procedural, de alto e baixo nível e, principalmente, padronizada.

A linguagem C disponibilizada pela NVIDIA foi estendida em algumas palavras-chaves e alguns construtores, os quais serão descritos mais tarde. A facilidade fornecida para o desenvolvedor de *software* tem como motivação empregar um mesmo ambiente de programação para ambas as arquiteturas de CPU e GPU.

O mesmo compilador C, o *nvcc*, possibilita que algoritmos seriais sejam compilados e executados no computador hospedeiro, ou *Host*, e também possibilita a compilação dos algoritmos definidos no programa para serem paralelos, que serão executados no dispositivo gráfico, ou GPU.

Tabela 5: Principais modelos da família da NVIDIA.

Modelo	Processadores	Relógio (núcleo)	Relógio (Shader)
GeForce 8400 GS	16	450 MHz	900 MHz
GeForce 8800 GT	112	600 MHz	1500 MHz
GeForce 8800 GTS	96	500 MHz	1200 MHz
GeForce 9600 GT	64	600 MHz	1625 MHz
GeForce 9800 GT	112	600 MHz	1800 MHz
GeForce 9800 GTX	128	675 MHz	1,688 MHz
Tesla C1060	240	1,3 GHz	800 MHz
GTX280	240	602 MHz	1296 MHz
GTX480	480	700 MHz	1401 MHz

O modelo escolhido *9800 GT* é compatível e escalável com modelos superiores existentes na família G80 e G92, Quadro, Tesla e a recente família lançada em 2008, GT 200, com 240 núcleos.

⁹ Mais placas de vídeo idênticas podem ser interligadas utilizando a tecnologia SLI, aumentando o Dispositivo de forma escalar.

4.7 Arquitetura da GPU G80

Uma GPU é normalmente construída em uma placa de vídeo de um computador, e seu acesso pelo *Host* ocorre por um barramento compartilhado de entrada e saída, PCI, PCIe (PCI Express) ou AGP. Atualmente, pode ser encontrada em versões simplificadas e integradas diretamente na própria placa-mãe do computador.

A GPU possui um grande conjunto de processadores (núcleos), atualmente na ordem de centenas, que podem acessar diretamente uma memória global existente na própria placa gráfica. Isto permite uma maior flexibilidade de programação do que as gerações anteriores de GPUs e facilita aos desenvolvedores a implementação do processamento paralelo através de funções chamadas “*kernel*”, acionada pelo computador *Host*, que serão descritos mais adiante.

A família *Geforce* é uma combinação de multiprocessadores independentes “*Streaming Multiprocessor*” (SM), onde cada um é equipado com oito processadores genéricos “*Streaming Processor*” (SP), os quais sempre executam a mesma operação, semelhante a uma unidade SIMD (KANTER, 2008).

A GPU 9800GT, por exemplo, consiste em quatorze SM. Estes 112 núcleos de processamento executam as instruções de um *kernel* a uma frequência de relógio (“*Processor Clock*”) de 600 MHz (NVIDIA, 2008).

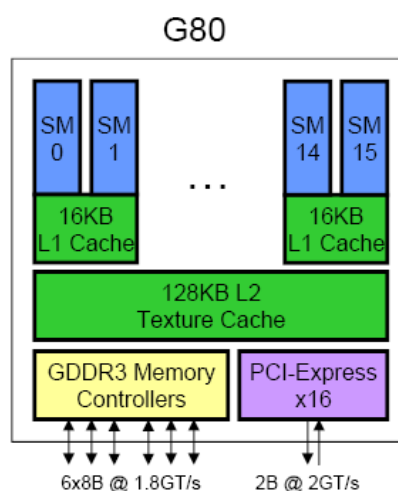


Figura 16: Arquitetura da família GeForce G80.

O multiprocessador utiliza estes processadores para executar as instruções em grupos de 32 elementos. Cada elemento é chamado de uma “*thread*” que são

pequenos processos. Os grupos de 32 *threads* são chamados de “*warps*”¹⁰. Uma “*Thread Processing Cluster*” (TPC) é formada por duas SMs (Figura 16) que compartilham um cache L1 de alto desempenho.

São encontradas em uma SP (Figura 17) oito unidades lógicas e aritméticas (ALUs de 32 bits) de ponto flutuante e precisão simples. Uma unidade aritmética¹¹ para soma e multiplicação de inteiros de 32 bits, operações de comparações, operações booleanas, (AND, OR e XOR), deslocamento de bit (bit-shifting) e computação de várias funções algébricas como, por exemplo, interpolação planar, trigonométrica, exponencial e funções logarítmicas.

Adicionalmente, as SP possuem duas unidades funcionais (SFU) que podem executar operações complexas como: multiplicações de ponto flutuante, cálculo de raiz quadrada, seno e cosseno, com baixa latência. Duas operações deste tipo podem ser executadas em um ciclo de relógio em todas as SP.

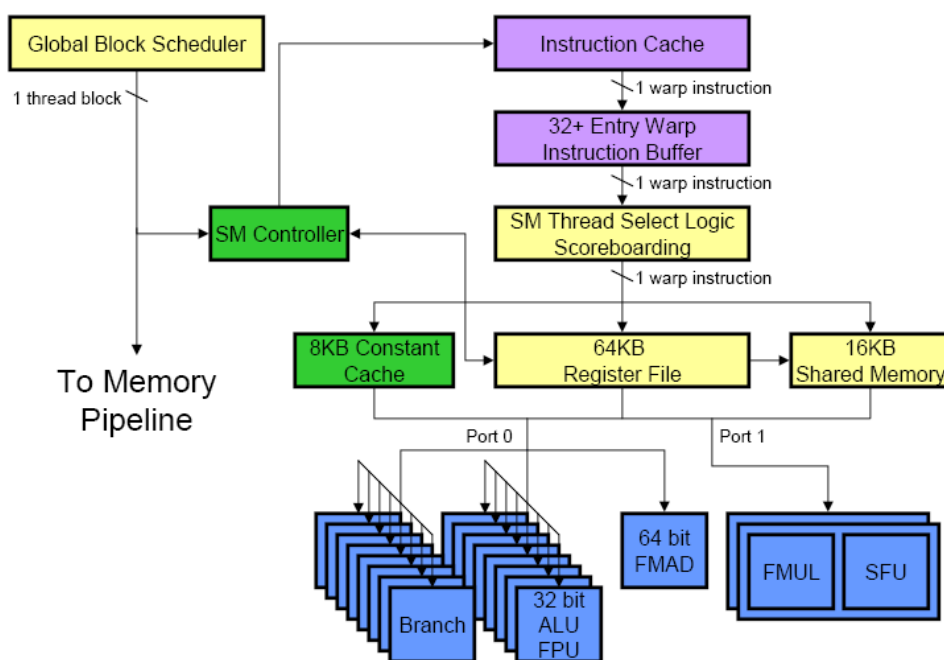


Figura 17: Arquitetura de uma *Streaming Processor* (SP).

As unidades aritméticas e SFUs são totalmente “*pipe line*”, permitindo que uma nova instrução entre em execução antes que uma instrução prévia tenha terminado, produzindo ~450 GFLOPs (14 SM * 18 FLOPs/SM * 1.8 GHz), desempenho de pico,

¹⁰ O multiprocessador SM leva quatro ciclos de relógio para executar uma *warp* de 32 *thread*'s: a mesma instrução é executada quatro vezes no mesmo processador SP para dados diferentes.

¹¹ O pedido de patente US 2008/0184211 aborda o conjunto de instruções utilizado pelo dispositivo GPU da NVIDIA.

para a GPU 9800 GT. Esta GPU ainda acessa a memória global com uma largura de banda de 86.4 GB/s.

4.8 Hierarquia de Memória

A hierarquia de memória foi projetada para permitir o compartilhamento de dados entre as *threads*, e reduzir o acesso à memória Global externa ao chip (NVIDIA, 2008). A Figura 18 apresenta um resumo dos cinco tipos de memórias existentes (relacionadas na Tabela 6) e disponíveis na arquitetura da GPU: quatro são construídas internamente na estrutura do chip e uma memória global acessada localmente na placa de vídeo, por uma via de 256 bits, externamente ao chip.

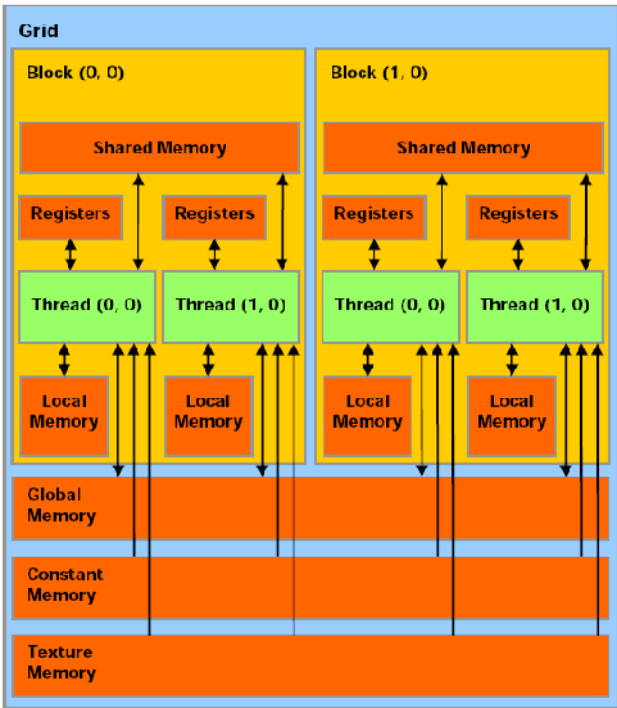


Figura 18: Hierarquia de memória de uma Geforce G80.

Tabela 6: Tipos de memória da arquitetura de memória GeForce G80.

Memória	Localização	Tamanho	Latência	Somente Leitura	Descrição
Registro	On-ship	8.192 registros de 32 bits	latência de Registro	Não	Cada multiprocessador SM possui acesso a 8.192 registradores de 32 bits: acesso local de baixa latência, é acessado somente pela <i>thread</i> e a informações nos registros existem enquanto a <i>thread</i> existir.

Global	Off-ship	até-768 MB	200-300 ciclos	Não	DRAM – Todos os dados são alocados nesta memória no início da execução do Kernel. É diretamente endereçável por ponteiros. Todas as <i>threads</i> acessam esta memória e as informações criadas existirão enquanto o kernel existir.
Compartilhada	On-chip	16 KB para cada SM	$\cong$ latencia de registro	Não	Memória local que pode ser acessada pelas <i>threads</i> de um bloco. É organizada em 16 bancos. Muitas vezes, é possível organizar o acesso das <i>threads</i> aos bancos de forma a evitar conflitos. As informações serão mantidas até o final do <i>thread</i> bloco.
Constante	On-chip cache	64 KB total	$\cong$ latencia de registro	Sim	8 KB de cache por SM, com os dados originalmente residentes na memória Global. O limite de 64KB é definido pelo modelo de programação. Frequentemente é utilizada para tabelas de pesquisas.
Textura	On-chip cache	Até a global	> 100 ciclos	Sim	Cache de 16 KB para os dois SM, com os dados originalmente residentes na memória Global. São úteis em certas aplicações de processamento de imagens.

4.8.1 Registrador local por *thread*

Na arquitetura da família G80 e G92, os multiprocessadores SM possuem 8.192 registradores (NVIDIA, 2008) de acesso local, com baixa latência. Este tipo de memória foi construída no próprio chip da GPU, com tecnologia SRAM. Estes são particionados dinamicamente entre as *threads* pelo compilador. Tais registros (BUCK e BASTIAAN, 2008) estão contidos internamente em bancos de arquivos logicamente divididos em raias, ou “lanes”. Cada banco possui números definidos de entrada, e cada entrada é constituída por uma palavra de 32 bits. Uma raia é designada para cada unidade de processamento SP, correspondentes a entradas em diferentes raias, e que podem ser populadas com diferentes *threads* executando o mesmo programa, facilitando assim a execução no modelo SIMD. O número de entradas em um registrador local (8.192) é suficientemente grande para suportar as múltiplas *threads* concorrentes que estão sendo executadas nas unidades de processamento. O período de latência destes bancos de arquivos de registros é assumido com sendo de dois ciclos de processamento na SP. As variáveis declaradas no *kernel* residem neste tipo de memória de acesso rápido.

4.8.2 Memória compartilhada por *bloco*

As *threads* que estão sendo processadas internamente na SM acessam também uma memória compartilhada *cache* L1, de escrita e leitura de 16KB, também construídas internamente no chip (NVIDIA, 2008). Estas são úteis para armazenar dados que podem ser compartilhados entre as *threads* do bloco, e trocar informações com baixa latência. Este tipo de memória é tão rápida quanto os registradores locais que as *threads* utilizam.

Cabe ao desenvolvedor¹² identificar a parte do código na qual haverá vários acessos de leitura e escrita por parte da *thread*, e se não for explícito no código o uso deste tipo de memória, o compilador irá assumir que o acesso será feito através da memória global. A memória compartilhada é cerca de 100 a 150 vezes mais rápida e o desenvolvedor poderá explicitar o uso desta memória através do código, como por exemplo:

```
__shared__ float shared[32];  
  
__device__ float device[32];  
  
shared[threadIdx.x] = device[threadIdx.x];
```

4.8.3 Memória Constante e Textura por *grid*

Na estrutura de memórias, é disponibilizada memória de acesso apenas de leitura para armazenamento de constantes. Estas variáveis são organizadas e escritas somente pelo *Host*. No caso de processamento de imagens que contém texturas de duas dimensões, é disponibilizada uma memória de textura do tipo somente leitura. Por serem memórias de acesso do tipo rápido, estas reduzem o período de latência do dispositivo e podem ser acessadas por todas as *threads* que estão sendo executadas na GPU. O uso desta memória é semelhante ao da memória compartilhada, só que esta é limitada apenas para leitura.

4.8.4 Memória Global por *grid*

O dispositivo acessa uma memória externa, randômica, ao chip da GPU (DDR3), inserida na arquitetura da placa de vídeo. Esta possui a finalidade de propósitos gerais. Todas as *threads* que estão sendo executadas no dispositivo podem acessar

¹² Algumas aplicações são engargaladas (bottleneck) pela banda disponível na memória Global da GPU, subutilizando o poder computacional de uma GPU.

este tipo de memória global, para leitura e escrita dos dados que estão sendo processados. Apesar deste tipo de memória disponibilizar amplo espaço (512 MB ou superior), ela é a memória que apresenta maior latência para o dispositivo, chegando a centenas de ciclos: a arquitetura não possui recurso de cache de dados para este tipo de memória.

4.8.5 Coalescência de memória

Memória global deverá, preferencialmente, possuir um comportamento organizado chamado de coalescência de memória (NVIDIA, 2008). É semelhante à leitura de uma linha inteira de dados de uma memória cache em oposição à leitura de apenas uma palavra desta cache.

Este comportamento está na definição dos tipos e estrutura de dados que as *threads* irão manipular. As *threads* possuem uma identificação, ou *id*, e são seqüencialmente numeradas de acordo com a dimensão x, y e z da *thread* no bloco. Estas são executadas em ordem, em conjuntos de 32 *threads* ou *warp*.

A coalescência é importante para manter a alta taxa de transferência entre a memória global e o dispositivo: ao acessar a memória, uma *warp* de 32 *threads* executará uma leitura ou escrita no modelo SIMD. O acesso à memória é feito em uma sequência de duas fases: metade de um *warp* de cada vez. Este executa uma operação de leitura na memória global ou “gather”, ou uma operação de escrita na memória ou “scatter”, em conjuntos consolidados de dados, o que aumenta o desempenho do dispositivo. Caso este processo de leitura seja feito por uma ou poucas *threads* da *warp*, este serviço será ineficiente e até individual, o que faz decair o desempenho do dispositivo.

Cada elemento de array deve estar alinhado e possuir 4, 8 ou 16 bytes. As *threads* na meia *warp* devem acessar o endereçamento de memória consecutivamente, na ordem. Por exemplo, a *thread* id n deve acessar o endereço de memória $HalfWarpBaseAddress^{13} + n$.

¹³ o endereço base da memória é do tipo `type*` descritos no Capítulo 6.1.2.1 (NVIDIA, 2008)

4.9 Grids de Threads

As *threads* são criadas em tempo de execução e na arquitetura G80 e G92 (NVIDIA, 2008). Estas são organizadas em três níveis hierárquicos: no nível mais alto, cada *kernel* cria um simples *grid* de *threads*, e este *grid* é dividido em vários *blocos* chamados de *threads blocos*, de acordo com a entrada de dados a serem processados.

O número máximo de *threads* por *bloco* na arquitetura do G80 e G92 é de 512 (NVIDIA, 2008). Cada *thread bloco* é escalonado para uma simples SM durante o período de execução, e cada *thread* recebe uma identificação única, ou *thread id*, assim como o *thread bloco* recebe uma identificação única, também, ou *bloco id*. Esta identificação é criada no tempo de execução e pode ser bidimensional para as *grids* e tridimensional para os *blocos*. É através desta identificação que a *thread* saberá qual parte dos dados alocados na memória deverá ser processada.

A função escolhida no algoritmo a ser processado na GPU é chamada de *kernel* e é esta função que será distribuída sincronamente nos diversos núcleos da GPU, através de *blocos*.

Esta abstração foi criada para organizar a distribuição destas funções em *grids*, compostas por *blocos de threads*. As *grids* são montadas a cada ciclo de um *kernel*, e não podem ser executadas em paralelo, mas pode haver um ou mais *kernel* a serem processados pela GPU. O processamento do *kernel* pela GPU não impede que outra função (função *serial*) seja executada no computador *Host*, ao mesmo tempo em que, às vezes, é conveniente este paralelismo.

Os parâmetros que especificam o número de *blocos* em cada dimensão é uma variável informada denominada *gridDim*, e são fornecidas em tempo de execução ao *kernel*: *gridDim.x* e *gridDim.y*, que especificam o número de *blocos* na dimensão x e na dimensão y, respectivamente. Os valores de *gridDim* podem ser um valor qualquer entre 1 a 65.536 (NVIDIA, 2008). Assim que o *kernel* é lançado para execução, estas dimensões não poderão mais ser alteradas. Todas as *threads* de um *bloco* compartilham o mesmo valor de *blockId*. Os valores de *blockId.x* variam entre 0 e *gridDim.x-1*, e os valores do *blockId.y* variam entre 0 e *gridDim.y-1*.

É possível exemplificar, através da Figura 19, o *kernel*₁. Este é definido e descrito no código fonte pelo desenvolvedor, o qual instrui a chamada do *kernel* com os

seguintes parâmetros: número de *threads* que cada *bloco* possui e número de *blocos* necessários para a execução de todos os valores de entrada. O *kernel* é lançado em tempo de execução, e a *grid*₁ criada consiste em seis *blocos*, no exemplo, organizados em um array de 2 x 3. Cada *bloco* no array é identificado pelo *blockId.x*, *blockId.y*. Neste exemplo, o *bloco*_(1,1) possui *blockId.x*=1 e *blockId.y*=1.

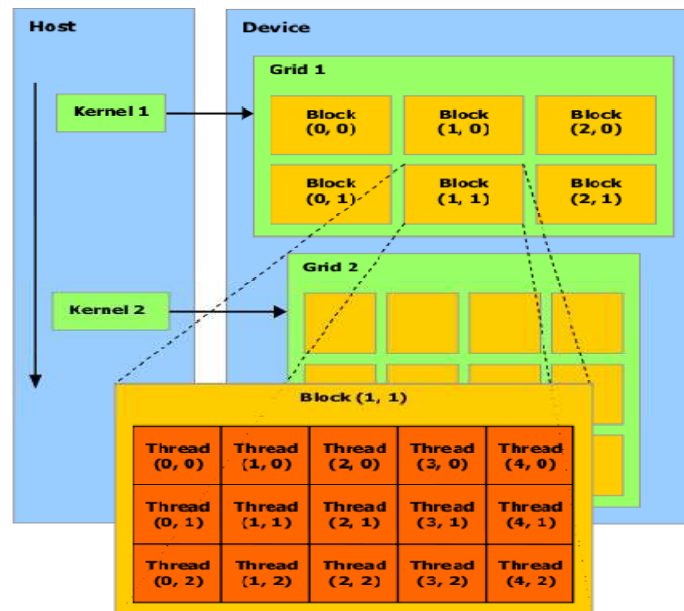


Figura 19: Abstração para distribuição das *threads*.

Todos os *blocos* pertencentes a um *grid* estão organizados em um array de *threads* com o máximo de três dimensões (NVIDIA, 2008). Todos os blocos da *grid* possuem a mesma dimensão. Cada *threadId* possui três coordenadas: x, y e z. Os parâmetros são *thread.x*, *thread.y* e *thread.z* consecutivamente. O número máximo¹⁴ de *threads* permitidas dimensionadas em um *bloco* é 512. Caso contrário um erro será lançado pelo dispositivo abortando a execução do *kernel*. Na Figura 19, são exemplificados seis *blocos* de quinze *threads* cada um, fornecendo um total de noventa *threads* para a execução do *kernel*. Cada *bloco* é organizado em um array de *threads* de tamanho 6 x 3 x 5. Na figura, as 15 *threads* do *bloco*_(1,1) são mostradas: *thread*_(0,1) possui as coordenadas *threadId.x*=0 e *threadId.y*=1.

As *threads* que estão no mesmo *bloco* podem compartilhar dados através da memória *cache* de 16 KB. Por estarem sendo executadas na mesma SM. Estas

¹⁴ Tamanho máximo de dimensão de um bloco: 512 x 512 x 64 thread's; Tamanho máximo de dimensão de um *grid*: 65.535 x 65.535 thread's

podem ser sincronizadas através de uma barreira de sincronização¹⁵. Para isto, basta o *kernel* invocar a primitiva “**__syncthreads**”.

Todos os *blocos* podem compartilhar a memória de acesso global, mas este acesso é mais lento que o acesso à memória *cache* compartilhada.

Finalmente, as *threads* são organizadas internamente em blocos chamados “*warps*”, com 32 *threads* para cada *warp*. Um *warp* é um conjunto de instruções que é executado tradicionalmente em arquiteturas SIMD distribuídas em quatro ciclos de relógio, nas oito SPs de uma SM (NVIDIA, 2008). Assim, os *warps* são formados em seções contínuas de oito *threads* dentro de uma *thread bloco*, e as primeiras 32 *threads* no *bloco* formam o primeiro *warp*, as 32 *threads* seguintes formam o segundo *warp* e etc.

Os multiprocessadores SM escalonam os *warps* intercaladamente, instrução por instrução básica do *kernel*. Este processo esconde a latência de acesso à memória global e também a longa latência existente nas operações aritméticas.

Uma tabela interna na GPU indica quando todos os operandos utilizados pelas *warps* estarão prontos para a execução. A SM então executa a mesma instrução para as 32 *threads* da *warp*. A SM distribui somente uma instrução por vez para todas as *threads* na *warp*.

Quando uma *thread* está sendo executada na *warp*, estas podem tomar caminhos diferentes. É assumido que múltiplos passos serão requeridos para completar a execução. Deste modo, esta execução será lenta na mesma medida em que cada *thread* executada tomar um caminho diferente. Desta feita, é altamente desejável evitar esta situação: esta se generalizará para o mesmo grupo de *threads*. Além disso, se o número de *threads* por cada *thread bloco* não for uniformemente divisível pelo tamanho da *warp*, qualquer resto desta *warp* será um recurso desperdiçado, ou seja, teremos núcleos sem o que fazer.

Quando uma *warp* entra em estado de espera ou “*stall*”, a SM providencia a troca desta por uma outra *warp* (NVIDIA, 2008) disponível do mesmo *thread bloco*, ou um *warp* pronto em uma outra *thread bloco* da SM. As SM entram em estado de espera, ou “*stalls*”, somente se não há *warp* com operandos prontos e disponíveis. A

¹⁵ *Thread's* são por outro lado independentes e sincronização cruzadas nos diferentes *thread bloco* podem ser seguras somente ao final da execução do *kernel*

arquitetura deste projeto favorece naturalmente as aplicações que possuem pouco controle divergente no *kernel*. Grandes regiões de dependência de dados são geralmente inadequadas para esta arquitetura.

4.10 API de Alto e Baixo Nível em uma GPU

Estes estudos focaram exclusivamente os aspectos computacionais da GPU da NVIDIA, especificamente CUDA, onde é fornecida uma API fortemente baseada na linguagem C/C++, de alto e baixo nível e de fácil aprendizagem (NVIDIA, 2008). O Kit¹⁶ de Desenvolvimento de Software ou “*Software Development Kit*” (SDK) é uma API de baixo nível o qual permite que o desenvolvedor configure os acessos à memória *cache* interna do dispositivo, sua memória *global*, número de *threads por bloco* e número de *blocos*.

4.11 CUDA

CUDA quer dizer “*Compute Unified Device Architecture*”, e é uma arquitetura de *hardware* e *software*¹⁷ desenvolvida para suprir e conduzir a computação na GPU como um dispositivo computacional de dados paralelos, sem a necessidade de mapear a API gráfica (NVIDIA, 2008). Em vez disso, a linguagem acessa diretamente os recursos nativos disponíveis no *hardware* gráfico.

4.11.1 Arquitetura de Pilha de Software

CUDA inclui duas interfaces de programação (API): uma de alto nível, ou “Runtime API”, e outra de baixo nível, ou “Driver API” (NVIDIA, 2008). O Runtime trabalha no topo do Driver, onde todas as chamadas Runtime são traduzidas e processadas em instruções simples pelo Driver, e duas bibliotecas matemáticas de alto nível de uso comum, CUFFT e CUBLAS: estas bibliotecas contêm subprogramas para Álgebra Linear Básica – BLAS e Transformada de Fourier Rápida - FFT.

O dispositivo processador gráfico (GPU) é habilitado a operar e disponibilizar recursos como sendo um co-processador para um processador central (*Host*). Nesta função, a GPU possui sua própria unidade de memória, assim como unidades de processamento que estão separadas do processador central (*Host*). Para que o co-

¹⁶ SDK foi disponibilizado a partir de 2007, gratuitamente (mas não é um open source).

¹⁷ CUDA, no momento, é disponibilizado apenas para a família GeForce, Quadro, Tesla, GT200 da NVIDIA.

processador realize um trabalho, os dados a serem processados deverão ser transferidos da memória do processador (*Host*) para a memória do dispositivo co-processador (GPU) através de um barramento serial.

A partir deste instante, será chamada de “*Host*” a unidade processadora central ou CPU e de “dispositivo” a unidade co-processadora ou GPU.

Como há a etapa de transferência de dados entre o *Host* e o *dispositivo*, é necessário e importante considerar o tempo (I/O) despendido nesta transferência de dados: a transferência de dados ocorre por uma conexão serial, porta padrão “PCI Express”, x16 ou v2.0. Esta porta serial, teoricamente, possui uma largura de banda de 16 GB/s (8 GB/s em cada direção) (NVIDIA, 2008).

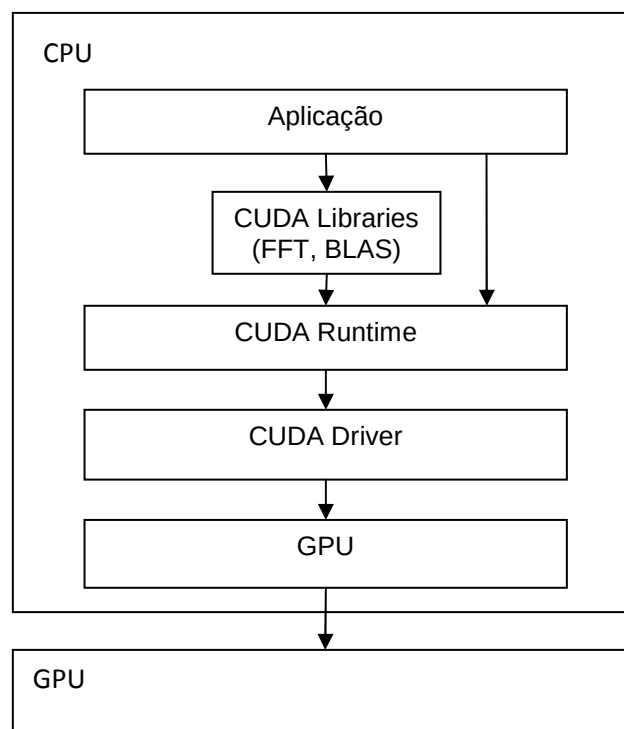


Figura 20: Pilha de *software* da arquitetura do SDK.

CUDA utiliza o desenvolvimento de *software* através da linguagem-padrão de programação C/C++, além das funções de bibliotecas específicas contendo extensões desta linguagem, de forma a gerar os objetos para o *dispositivo*.

A API é disponível para os *dispositivos* da série Geforce 80 e 92, família Quadro e a recente família GT 200, disponibilizada em 2008. A NVIDIA tem informado que adota a política de compatibilidade da tecnologia para os futuros *dispositivos*.

A API CUDA é multiplataforma, disponibilizada atualmente para os sistemas operacionais Microsoft Windows, MAC OS X e Linux (NVIDIA, 2008).

4.11.2 C/C++ Linguagem para GPU

CUDA é um modelo de programação ANSI C (NVIDIA, 2008) estendido por construtores e palavras-chaves definidas para soluções de processamento paralelo, onde não é necessário qualquer conhecimento sobre programação gráfica como o OpenGL ou DirectX. É disponibilizado em um ambiente único para o desenvolvimento de aplicações em ambas as plataformas: *Host* (CPU) e *dispositivo* (GPU).

Partes do código-fonte são escritas e específicas para serem compiladas, tanto para o *Host* quanto para o *dispositivo*. Funções, objetos e variáveis descritos em único código fonte também poderão ser especificados através de declaradores para cada arquitetura alvo.

CUDA fornece uma ferramenta de desenvolvimento de *software* (SDK) introduzindo classes de abstrações de como o *hardware* se comporta, facilitando o desenvolvimento da aplicação, assim como a manter a compatibilidade do código com futuras gerações de Dispositivos.

4.11.3 Compilador

Passos adicionais foram introduzidos ao compilar um programa-fonte utilizando o compilador *nvcc* desenvolvido pela NVIDIA para este ambiente. Isto porque o programa alvo possui duas arquiteturas de processadores distintos: uma para o *Host* e outro para o *dispositivo*.

O primeiro passo ao compilar o programa é separar o código fonte para cada arquitetura alvo (MURPHY, 2008). Este processo é realizado pelo pré-processador nomeado de *Cudafe* baseado no “Edson Design Group” – EDG - (HALFHILL, 2008).

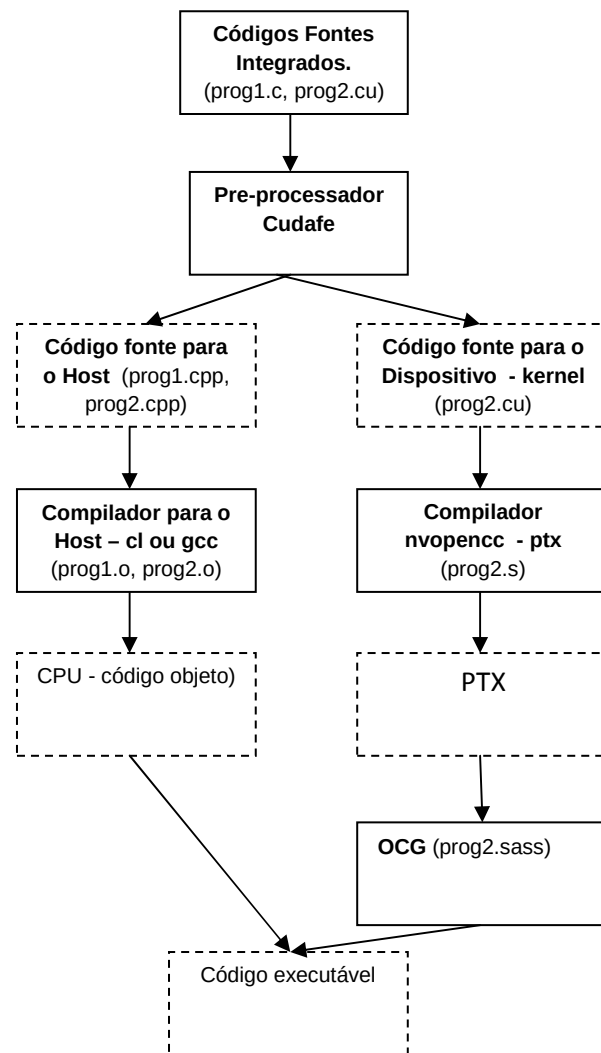


Figura 21: Fluxo do processo de compilação - *nvcc*.

O pré-processador *Cudafe* analisa gramaticalmente o código do programa fonte (MURPHY, 2008) e divide este em dois diferentes arquivos, um para cada arquitetura alvo:

- Para a arquitetura do *Host*, *Cudafe* cria o arquivo fonte padrão “.cpp”, ficando pronto para ser compilado. Este poderá ser compilado pelo compilador C/C++ da Microsoft (cl) ou pelo compilador GNU C/C++ do Linux (gcc);
- Para a arquitetura do *dispositivo*, *Cudafe* cria diferentes conjuntos de arquivos “.cpp”. Estes arquivos fontes são então compilados pelo compilador *nvopencc*, baseado no “open source” Open64¹⁸.

¹⁸ Inicialmente desenvolvido para arquitetura Intel Itanium

- O compilador *nvopencc* gera arquivo “.ptx” que está no formato de linguagem *assembly* do dispositivo, chamado de “*Parallel Thread eXecution*”;
- Os arquivos *ptx* são passados pelo “Optimized Code Generator” (OCG), onde são manipuladas as alocações de registros, com o escalonamento das instruções de acordo com o *hardware* particular da placa gráfica que está sendo utilizado. Estes arquivos *ptx* são compilados em tempo de execução “Just in time” (JIT) para o código binário do Dispositivo.

4.11.3.1 Opções do Compilador *nvcc*

O compilador *nvcc* suporta algumas opções (RYOO *et al.*, 2007) que o desenvolvedor poderá utilizar para depurar a função descrita no *kernel*: para aumentar o desempenho deste ou para depurar erro de programação. Três *flags* são especiais para isso: “*-ptx*”, “*-ptxas-options=-v*” e “*-cubin*”, onde o compilador *nvcc* disponibiliza arquivos de saída com formato texto e é legível para o desenvolvedor.

- O *flag* “*-ptx*” ou “*-keep*” fornece a inspeção do código *assembly ptx*, e informa como o compilador interpretou os parâmetros de entrada e como podem ser otimizados manualmente para aumentar o desempenho do *dispositivo*, como exemplo: manipular o “*loop unrolling*” (*ptxas-options=-v*); reportar o local utilizado pela memória (Imen).
- O *flag* “*-cubin*” informa os recursos a serem utilizados pelo código do *kernel* que será executado no *dispositivo*, incluído a quantidade de memória a ser utilizada por cada *thread-bloco* e o número de registros utilizados por cada *thread*.

4.11.4 Extensões da Linguagem

A parte do código-fonte que será executada é chamada de *kernel*, e é uma função chamada através do *Host* para ser executada no *dispositivo*. É identificada no código através (BUCK e BASTIAAN, 2008) da sintaxe “<<<>>>”. Quando uma função ou

kernel é chamada, a seguinte extensão de linguagem é utilizada: $\langle\langle\langle n, m \rangle\rangle\rangle$: a variável n representa o número de *blocos* e m o número de *threads por bloco*¹⁹.

Esta notação “ $\langle\langle\langle \rangle\rangle\rangle$ ” foi selecionada pelo fato de que não é utilizada na linguagem de programação C/C++ (BUCK e BASTIAAN, 2008). Esta extensão é fornecida entre o nome da função e os argumentos/parâmetros desta, de forma a fornecer o metadado necessário para o compilador *nvcc*. Por exemplo, a função (f) possui os parâmetros (a) e é chamada utilizando a seguinte sintaxe: $f \langle\langle\langle n, m \rangle\rangle\rangle(a)$. Neste caso, $n \times m$ cópias da função descrita no *kernel* serão executadas nos *cores* do dispositivo.

Como mencionado na Seção 4.9, a organização de uma *grid* é determinada pelos parâmetros, descritos acima, especiais n e m . Estes são declarados no instante inicial de execução do *kernel*. O primeiro parâmetro n do *kernel* f especifica a dimensão do *grid* em função do número de *blocos*, e a segunda dimensão m especifica a dimensão de cada *bloco* em função do *número de threads* (BUCK e BASTIAAN, 2008).

Os parâmetros n e m são estruturas que podem conter uma, duas ou três dimensões e sua apresentação (NVIDIA, 2008) está relacionada à estrutura de dados do problema que o algoritmo irá tratar. *Grids* possuem uma estrutura de array em duas dimensões de blocos (a terceira dimensão é ignorada). Por exemplo, para os parâmetros $nGrids$ (8,8,8); $nBlocks$ (512,1,1), são definidos na chamada da “função do kernel”: $f \langle\langle\langle nGrids, nBlocks \rangle\rangle\rangle(a)$; É possível observar que estes parâmetros não precisam ser constantes, e podem ser passados ao *kernel* em tempo de execução por variáveis do sistema, e este argumento é, neste estudo, utilizado como parâmetro para a estrutura de dados de entrada (dados do problema) e para a estrutura de dados de saída (solução do problema) no processamento da simulação do problema *Job Shop*.

Aplicações seriais, ótimas candidatas, que podem ser transcritas ou dotadas para modelo paralelo, através da construção de *kernel*, são as funções que possuem entradas e saídas independentes e são chamadas muitas vezes. Por exemplo, operações de adição, multiplicação ou *saxpy* utilizando um array de dados, onde cada subconjunto de elementos do array é processado por uma *thread* distinta.

¹⁹ Nas Patentes (BUCK, 2008) e (NICKOLLS, 2008), a NVIDIA denota a expressão “*bloco*” (NVIDIA, 2008) por “Cooperative Thread’s Array” (CTA).

No *código-fonte*, onde a aplicação é programada, haverá duas principais declarações (“decl spec”) para modificadores de função, as quais serão compiladas para o *dispositivo*: a extensão “__global__” e “__device__”. Quando a declaração para modificador de função “__global__” é encontrada no *programa fonte*, o compilador *nvcc* traduz o *código-fonte* em binário para ser executado no *dispositivo*, como descrito anteriormente, e o algoritmo ali contido é chamado de *kernel*. As instruções deste *kernel* serão executadas seqüencialmente, através de *threads* (BUCK e BASTIAAN, 2008).

A declaração para modificador de função global (BUCK e BASTIAAN, 2008), como o exemplo abaixo, é aplicada a uma função (*f*), passando como argumento um inteiro *a*, como descrito abaixo:

```
__global__ void f (int a) {  
    Instruções....  
}
```

Quando a declaração para modificador de função “__device__” é chamada (BUCK e BASTIAAN, 2008), o compilador *nvcc* traduz este código para ser executado no *dispositivo*, mas esta função somente poderá ser chamada por outra função executada no *dispositivo*. Assim como a especificação anterior, é possível exemplificar uma chamada a uma função (*g*), passando um argumento inteiro *b*, como descrito logo abaixo:

```
__device__ void g (int b) {  
    Instruções....  
}
```

Estas funções não possuem argumento de retorno, e são sempre declaradas com o tipo *void*, uma vez que estas serão executadas no *dispositivo*, e não no *Host*. Os argumentos de retorno são declarados nos parâmetros da função, e seus tipos deverão ser especificados explicitamente.

Não é permitida a declaração de variáveis estáticas internas nestas funções. E ainda, estas funções não podem ser recursivas, o que pode ser resolvido introduzindo uma função recursiva manual, com a manipulação de pilhas (BUCK e

BASTIAAN, 2008). Também o cálculo para pontos flutuantes é suportado somente para precisão simples.

Outra declaração para modificador de função “__host__” define que o código a ser compilado será executado no *Host*. Por *default*, todas as funções do *programa fonte* serão entendidas para serem compiladas e executadas no *Host*, mas a declaração “__host__” é útil quando aplicada à declaração de uma função que deverá ser compilada e executada tanto para o *dispositivo* quanto para o *Host*, exemplo descrito abaixo para a função *k*

```
__host__ __device__ void k (int c) {  
    Instruções....  
}
```

A função *k* é chamada em ambos ambientes: através do código do *Host* ou do *dispositivo*. Assim, o código é compilado duas vezes para ambas as plataformas.

As declarações para objetos de memória (BUCK e BASTIAAN, 2008), ou modificadores de variáveis, serão fornecidas para identificar a localização destes objetos/variáveis de memória. Exemplos de declaração de objetos: “global”, “device”, “shared”, “local”, ou “constant”.

Declaração de objetos de memória como do tipo “global” ou “device” poderão ser endereçáveis diretamente pelo código executado no *dispositivo* (BUCK e BASTIAAN, 2008). Declaração de objetos de memória *global* indica que o objeto/variável reside na memória do *dispositivo*, e é diretamente endereçável pelo *Host*. Deste modo, os objetos/variáveis podem ser acessados pelo *dispositivo* e pelo *Host*. Por exemplo, as declarações de objetos de memória global são aplicadas a objetos/variáveis (“a”) como __global__ int a.

A declaração de objetos do tipo “device” indica que o objeto/variável reside na memória do *dispositivo*, e não é diretamente endereçada pelo *Host*. Seguindo exemplo semelhante, os objetos de memória “device” são declarados como __device__ int a. Lembramos que as memórias do tipo *global* são mais custosas que as memórias do tipo *device*.

A declaração para memória compartilhada “shared” é específica para a *cache* compartilhada existente em cada *bloco*, e é acessada por todas as *threads* do *bloco*

através de um “*crossbar swich*”. Como exemplo, a declaração aplicada para o objeto variável (“x”) é descrita como `__shared__ int x`. A memória compartilhada é alocada no instante da criação de um *bloco*, e é utilizada por este até a última *thread* deste *bloco*. Memórias compartilhadas não podem ser acessadas por *threads* de *blocos* diferentes.

A declaração para memória compartilhada “*local*” é específica para os registradores acessados diretamente pelas *threads* do *bloco*. Como exemplo, a declaração aplicada ao objeto (“p”) é descrito como `__local__ int p`. O objeto é instanciado na criação da *thread*, e esta existirá enquanto a *thread* existir (BUCK e BASTIAAN, 2008).

A declaração de objeto de memória “*constant*” é específica para objetos residentes na memória do tipo “*somente leitura*”, existente internamente no *dispositivo*. Este espaço de memória é otimizado para ter um acesso rápido. A declaração deste tipo de memória (“a”) é descrito como `__constant__ int a`.

4.11.5 Variáveis Globais

- Dimensão da *grid*: *gridDim* – variável do tipo dim3^{20} (NVIDIA, 2008).
- Dimensão do *bloco*: *blockDim* – variável do tipo `dim3`.
- Índice do *bloco* interno na *grid*: *blockIdx* – variável do tipo `uint3`.
- Índice da *thread* interno no *bloco*: *threadIdx* – variável do tipo `uint3`.
- Tamanho da *warp* em uma *thread*: *warpSize* –variável do tipo `int`.

4.11.6 Função de Sincronização

Sincroniza todas as *threads* internas em um bloco: `__syncthreads()` (NVIDIA, 2008).

4.11.7 Função Atômica

Funções atômicas realizam operações de *leitura-modifica-escreve* uma palavra de 32 ou 64 bits residentes na memória *global* ou compartilhada, garantindo que outra *thread* não interfira na operação, exemplo (há outras operações): `atomicAdd()` que lê uma palavra de 32 bits, adiciona um inteiro e escreve na mesma posição da memória global ou compartilhada (NVIDIA, 2008).

²⁰ É um vetor do tipo inteiro que é usado para especificar dimensões baseado nos tipos de vetores `uint3` do tipo “Built-in”, construídos em tempo de execução.

4.11.8 Gerenciamento de Memória

cudaMalloc() – Alocação de Memória no Dispositivo

cudaFree() – Liberação de Memória no Dispositivo

cudaMemcpy() – Copia uma variável de memória do Dispositivo para o *Host* ou do *Host* para o Dispositivo.

Exemplo 1: /*copia valores de memória do *Host* para o Dispositivo*/

```
cudaMemcpy(a_device,a_host,size,cudaMemcpyHostToDevice);
```

Exemplo 2: /*copia valores de memória do Dispositivo para o *Host* */

```
cudaMemcpy(a_device,a_host,size,cudaMemcpyDeviceToHost);
```

É possível demonstrar, através de exemplo, como um algoritmo *serial* “somar vetores” é executado normalmente, e programado em uma arquitetura *single-core* ou *Host*: o algoritmo, para ser executado em um *dispositivo many-core*, o aninhamento do comando “**for**” é substituído pela distribuição dos índices da *grid* no processo de soma de vetores (NVIDIA, 2008).

Host:

```
void incrementaArrayNoHost(float *a, float *b, float *c, int N) {  
    int i;  
    for (i=0; i < N; i++)  
        c[i] = a[i] + b[i];  
}
```

Kernel:

```
__global__ void incrementaArrayNoDispositivo(float *a, float *b, float *c, int N) {  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    if (idx<N) c[idx] = a[idx] + b[idx];  
}
```

A chamada ao kernel é feita pela declaração:

```
incrementaArrayNoDispositivo <<< nBlocks, blockSize >>> (a, b, c, N);
```

Capítulo 5

Simulador *web* - arquitetura

5.1 Introdução

O Simulador em ambiente *web* dedicado ao estudo de algoritmos paralelos descrito na presente dissertação pode ser empregado em aplicações industriais ou acadêmicas (protótipos de pesquisa), fomentando o desenvolvimento de novas soluções ao difícil problema JSP.

Este Simulador aborda a resolução do problema JSP através do uso de uma arquitetura paralela de baixo custo disponível no mercado, a qual é integrada a um servidor de aplicação *web*. Deste modo, é possível através deste recurso de processamento paralelo criar novos métodos e acelerar o desempenho de uma solução JSP em torno de duas ordens de grandezas.

A versão do Simulador proposto permite escalonar até 100 *tarefas* para 100 *máquinas* em um problema JSP, apresentando os resultados do escalonamento em um modo gráfico, colorido e de fácil entendimento, inserido em um formato de documento portátil ou “*Portable Document Format*” (PDF). Este é um padrão aberto de documentação, o que permite abrir diretamente no navegador e possui diversas funcionalidades úteis, como por exemplo, rotacionar, ampliar, imprimir e salvar em disco o referido documento.

O sistema permite a inclusão de seqüências de tarefas individualmente para cada máquina, através de páginas dinâmicas “*Java Server Pages*”. Opcionalmente o usuário pode fazer também *upload* de novos problemas JSP ou fazer uso dos problemas já conhecidos, disponibilizados na literatura, como os problemas da *OR-Library*.

5.2 Motivação

Um ambiente adequado para a construção de novos métodos é a contribuição que este trabalho pretende fornecer. O modelo do Simulador, descrito na presente

dissertação, irá facilitar a identificação de métodos mais eficientes para resolução do difícil e importante problema “*Job Shop*”.

O protótipo do Simulador, desenvolvido em laboratório, utiliza uma heurística inspirada na *Busca Tabu*, a qual permite abordagem dos recursos computacionais da arquitetura paralela existente no Simulador. O algoritmo desenvolvido restringe-se ao escalonamento de tarefas ou *schedulling* para o problema JSP e foi descrito em detalhes no Capítulo 3.

O objetivo principal do Simulador é disponibilizar uma nova ferramenta de pesquisa para o desenvolvimento de heurísticas que fazem uso da arquitetura paralela, descrita neste capítulo.

Outro objetivo é disponibilizar ao pesquisador um *ambiente integrado* para desenvolvimento de *software* ou *IDE (Integrated Development Environment)*. Este *IDE* é configurado adequadamente para possibilitar ao pesquisador adicionar seu próprio algoritmo utilizando as interfaces e bibliotecas existentes no Simulador, isto é, o pesquisador gerar seu código neste ambiente de desenvolvimento inserindo seu algoritmo no módulo *kernel* da arquitetura para construir novos métodos para o problema “*Job Shop*”, utilizando as *templates* ali existentes.

Neste ambiente integrado foi utilizado a IDE Netbeans, “*open source*”, para integrar as diferentes camadas de *software* e, assim, utilizar três compiladores (*Java*, *gcc* e *nvcc*) no mesmo ambiente, facilitando o processo de compilação e *debug* do programa fonte. Testes poderão ser realizados assim como comparações com as heurísticas já construídas, utilizando as interfaces do sistema. O processo para fazer um novo método reconhecível pelo Simulador consiste de duas etapas:

- preparação do algoritmo que executa o *kernel* através da programação, compilação e *debugging* no ambiente de desenvolvimento;
- inserção do código, através de uma biblioteca dinâmica, no Simulador.

As soluções ou métodos descritos acima serão inseridas em bibliotecas “*.so*” (*shared object*) no diretório */lib* do Linux de modo que os resultados e soluções possam ser compartilhados e comparados através do Simulador. Pode-se pensar em bibliotecas como um conjunto de instruções realizando métodos diferentes que

dizem ao dispositivo GPU como lidar com diferentes tipos de dados de um problema JSP.

Através desta metodologia o pesquisador pode construir seu método no seu ambiente de desenvolvimento utilizando as *templates* existentes no Simulador e fazer *upload* de seu objeto ao servidor de aplicação e testar a eficiência de sua heurística em um laboratório. Os resultados poderão ser medidos, comparados e a solução será compartilhada com soluções de outros pesquisadores que partilham do mesmo ambiente.

5.3 Arquitetura do Simulador

A arquitetura do Simulador apresenta dois grandes desafios tendo em vista as características do problema JSP, os quais são: número de usuários distribuídos e o tempo de resposta do sistema. Esta arquitetura foi desenvolvida em camadas para suportar a demanda de requisições, facilitar sua manutenção e expansibilidade. Esta arquitetura também possui uma interface de entrada para os parâmetros do problema JSP, uma camada de serviços, e uma camada de processamento do problema através de uma arquitetura específica para processamento paralelo.

A arquitetura do Simulador utiliza tecnologia de objetos distribuídos para simular o problema do “*Job Shop*” (JSP) em um ambiente paralelo. O Simulador utiliza várias tecnologias emergentes como CUDA “*Compute Unified Device Architecture*” descrita no Capítulo 4.

Em princípio, este Simulador disponibiliza um ambiente para estudos e desenvolvimento de algoritmos paralelos utilizando a arquitetura CUDA da empresa fabricante de placas de vídeo NVIDIA. Este Simulador utiliza o modelo GTX 9800 da NVIDIA com 112 processadores, mas poderá ser facilmente expandido, também a baixo custo, para 240 processadores ou mais: pode-se utilizar a tecnologia “*Scalable Link Interface*” (SLI) da própria NVIDIA que possibilita a utilização de mais de uma placa gráfica no mesmo ambiente de *hardware*.

Novos modelos já estão disponíveis no mercado que utilizam a tecnologia CUDA, como a família GT200 com 240 núcleos ou a família Tesla com até 4 x 240 núcleos (KANTER, 2008).

Um ambiente construído com a arquitetura apresentada nesta dissertação foi instalado no laboratório de robótica CONTROLAB/NCE da UFRJ e por tratar-se de um estudo acadêmico não possui integração com banco de dados externo ou outros sistemas. No entanto, pode ser adicionado sem dificuldades na arquitetura ora proposta.

Esta arquitetura permite também que os usuários, com acesso à rede intranet, possam utilizar a mesma biblioteca de métodos para o problema “*Job Shop*” disponível no Simulador sem realizar *download* de outros programas ou reescrever linhas de comando de programação para obter uma solução ao problema JSP.

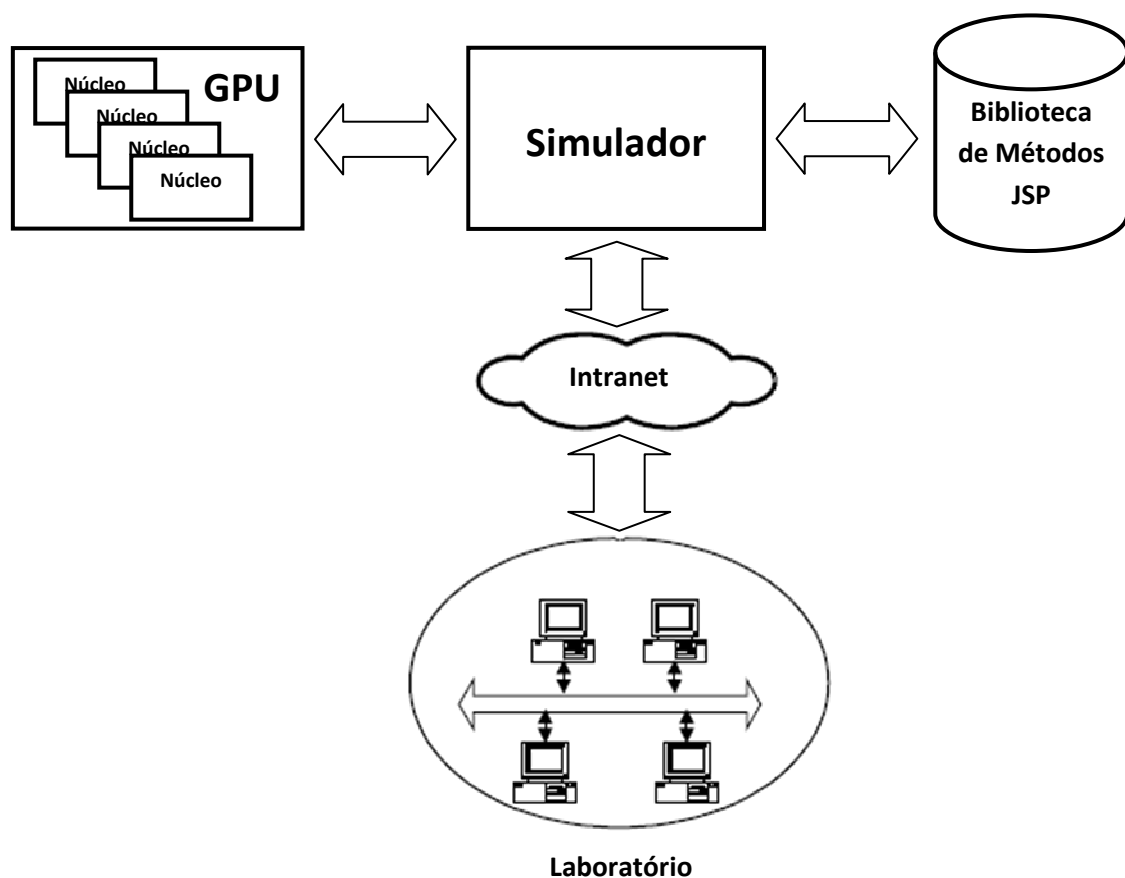


Figura 22: Arquitetura do Simulador *web* para problemas JSP.

A proposta pode ser melhor compreendida através da Figura 22, onde pesquisadores em um laboratório constroem métodos utilizando os recursos computacionais da computação paralela encontradas na arquitetura do Simulador, simulam e comparam este com os outros métodos ali existentes, através de uma biblioteca de métodos, de forma que outros pesquisadores do laboratório ou em

outros laboratórios, todos interligados pela *web*, possam compartilhar e evoluir na melhor resolução do problema JSP.

Os resultados da simulação são armazenados em um documento PDF o qual poderá ser feito *download* e armazenado (em disco) no sistema de arquivos do sistema operacional do cliente (máquina do pesquisador). Na estrutura deste documento PDF há um *link* para realizar o *download* da solução vista no correspondente documento em formato de arquivo XML, armazenado no servidor *web*. Este arquivo é armazenado em arquivo de *log* no *servidor de aplicação* para, posteriormente, ser recuperado e comparado com outros escalonamentos gerados através de outros métodos.

5.4 Arquitetura em Camadas

A estrutura do Simulador é baseada nas especificações *Java Edição Empresarial* ou “*Java 2 Enterprise Edition*” (J2EE), muito empregado na tecnologia *web*. A plataforma J2EE fornece funcionalidades que utilizam *software Java* distribuído, tolerante a falhas e multicamadas. O processamento do problema “*Job Sop*” e os recursos computacionais desenvolvem-se apenas no lado do servidor ou “*Server-side*”. Estes recursos são particionados em três camadas em termos da lógica da aplicação. Cada camada possui diferentes responsabilidades e estratégias definidas e estas possuem seus próprios componentes que interagem através de interfaces.

As camadas desta arquitetura são as seguintes:

- **Camada de Apresentação** contém componentes que tratam da *interface* com o usuário e sua interação. A camada de apresentação em uma aplicação *web* são as páginas HTML, páginas dinâmicas ou *Java Server Pages* (JSP), descritas anteriormente, definindo um cliente magro ou “*thin client*”. Algumas funções em “*Java Script*” são utilizadas nas páginas do Simulador para realizar validações de dados, como o número máximo de tarefas ou de *robôs*, por exemplo, assim como os estilos e fontes são definidos por uma linguagem de estilo “*Cascading Style Sheets*” (CSS), separando o formato do conteúdo no documento apresentado nas páginas HTML.

- **Camada de lógica de negócio** é compreendida por um servidor *web* e um servidor de aplicação. O servidor *web* possui componentes que trabalham em conjunto para modelar e apresentar as resoluções ao problema JSP. Estes componentes são *Servlets* e *JavaBeans*.
- **Camada de dados** compreende o processamento dos dados do problema JSP através de um dispositivo co-processador, “*General Purpose Graphics Processing Unit*” (GPGPU), visto no Capítulo 4. Os dados do problema são transferidos entre as duas plataformas de *hardware*, entre o *Host*, processador utilizado pelo *servidor de aplicação*, e o *dispositivo* paralelo GPU – co-processador existente na placa Gráfica – através de uma interface *Java Native Access* (JNA). A comunicação será feita através de troca de mensagens entre estes dois ambientes de *hardware* utilizando o padrão de arquivo XML.

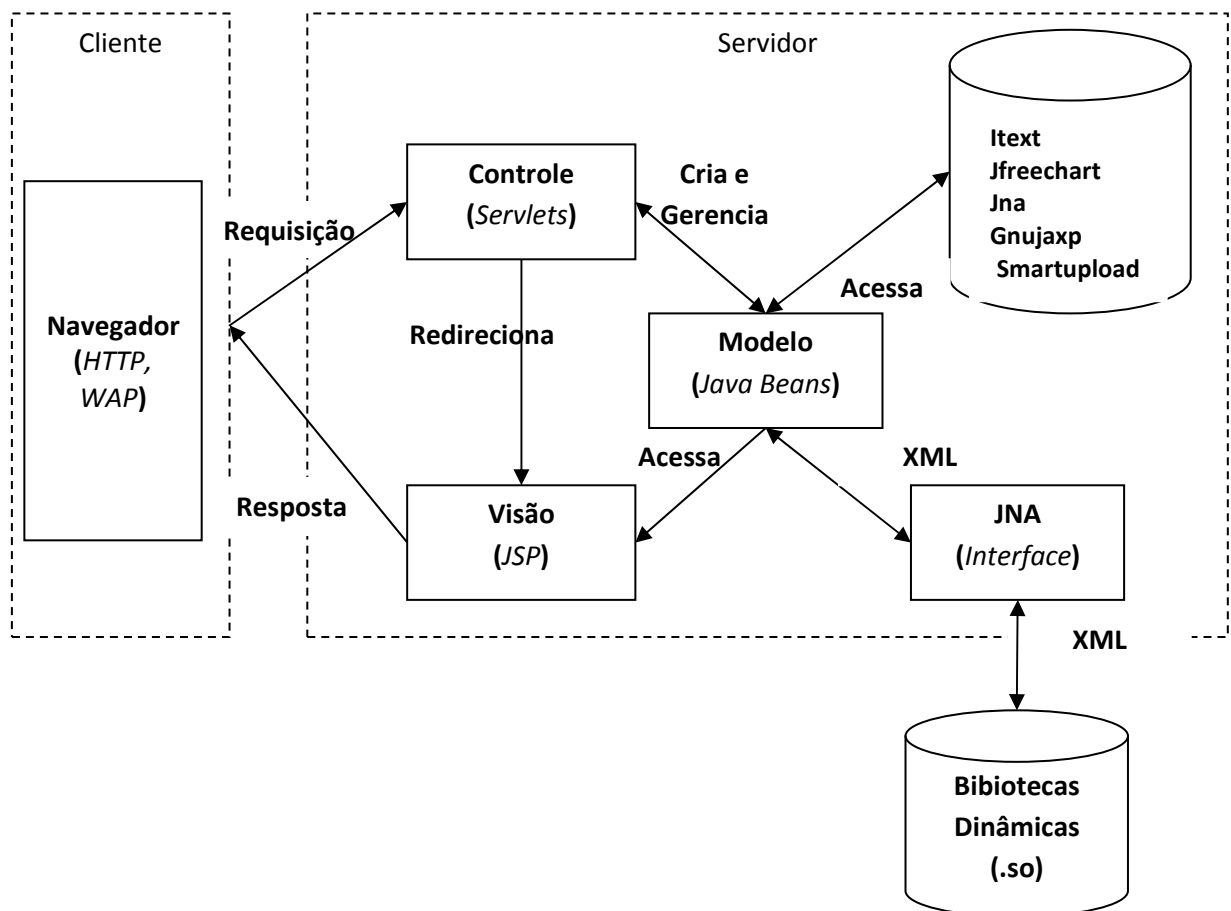


Figura 23: Arquitetura do Servidor de Métodos.

Uma das vantagens em particionar a aplicação em camadas é diminuir a dependência lógica entre estas camadas, diminuindo o custo de manutenção do sistema com a introdução de futuras funcionalidades. A arquitetura de três camadas pode ser vista através do modelo, visão e controle ou “*Model-View-Controller*” ou **MVC** (Figura 23), seguindo técnicas de padrões de projeto ou “*Design Patterns*”.

Como exemplo, é possível alterar a camada de apresentação com o mínimo de impacto na camada lógica ou na camada dos dados. Novas páginas podem ser inseridas com a especificação de outros métodos de resolução JSP, por exemplo: parâmetros de temperatura do método recozimento simulado (SA); número de população para o algoritmo genético (GA); e etc. Estas alterações não trariam impacto ao projeto e seria necessária apenas a introdução destes novos parâmetros na camada de negócio.

5.4.1 Servidor web

Foi utilizado o *Tomcat*²¹ como servidor web (*pure Java*), desenvolvido pela *Apache Software Foundation*. O *Tomcat* é um *servlet container*, ou seja, é um servidor onde são instaladas *Servlet's* e JSP (“Java Server Pages”) que são módulos que ampliam as funcionalidades do servidor. O *Tomcat* é também um “open source” multiplataforma de fácil instalação e configuração.

O *Tomcat* foi totalmente escrito na linguagem *Java* e esta arquitetura herdou as suas principais vantagens:

- **Portabilidade:** o Simulador aqui descrito pode ser implantado em várias plataformas, como por exemplo, Linux, Windows, Macintosh, Unix, sem a necessidade em modificar ou mesmo reconstruir a aplicação;
- **Facilidade de programação:** a programação é orientada a objetos, simplificando o desenvolvimento do Simulador. Outra facilidade no uso desta linguagem é o gerenciamento automático de memória: o desenvolvedor não precisa se preocupar com este gerenciamento. Este é gerenciado pelo *Java Virtual Machine*;

²¹ <http://tomcat.apache.org/>

- **Flexibilidade:** a linguagem *Java* é a mais difundida no momento, contando com uma enorme comunidade de desenvolvedores, ampla documentação e diversas bibliotecas e códigos prontos, dos quais o desenvolvedor pode usufruir.

5.4.2 Java Server Pages

“Java Server Pages” (JSP – mesmo acrônimo do “*Job Shop Problem*”) é uma tecnologia *Java* para geração de conteúdo: é um código HTML com *tags* que indicam quais partes são código *Java* utilizados para a montagem da página HTML. Um JSP é compilado dinamicamente pelo servidor de aplicação e convertido em *Servlet* para serem prontamente executados por este servidor. A grande vantagem do JSP está na velocidade, uma vez que o código *Java* fica compilado no servidor, ou seja, não é necessário ser interpretado toda vez que for utilizado.

O navegador (cliente) interpreta as páginas HTML. Estes são oriundos da compilação das páginas JSP, possibilitando ao usuário (pesquisador) a visualização do seu conteúdo. Assim, na camada de apresentação, o usuário interage com o sistema na construção de uma instância de problema do “*Job Shop*”: este visualiza no navegador (através de uma página HTML) as instruções descritas nesta página e preenche as caixas de diálogo com as informações solicitadas, interagindo passo a passo na descrição de cada tarefa e cada operação por robô. Com o uso desta metodologia, além de facilitar a visualização do problema que está sendo construído também há a diminuição dos erros de preenchimento. Quando há erros, a correção é fácil de ser feita: simplesmente voltando à página anterior, mantida no histórico do navegador.

Utiliza-se, também, páginas dinâmicas para a apresentação de erros ao usuário, assim como página de *uploads* de arquivos para os problemas já conhecidos, como os problemas descritos na *OR-Library*²².

²² OR-Library Brunel University – West London -.URL: <http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt>

5.4.3 Classe Servlet

Na versão básica do Simulador apresentado nesta dissertação foi necessária apenas a construção de uma classe *Servlet*. Esta é dinâmica e responsável pelas requisições/respostas (“*request / response*”) HTTP às páginas HTML do sistema. Este procedimento de interação com os usuários constrói os dados de entrada para o problema “Job Shop”.

As páginas HTML interpretadas pelo navegador mostram informações ao usuário e armazenam os dados do problema “Job Shop” no seu histórico, todas descritas através de *tags* HTML, como o número de tarefas, número de robôs, descrição de cada operação por tarefa, nome da instância, e-mail do usuário e etc.

Foi utilizado o método *doPost*, invocado no *servlet* do servidor de aplicação (uma extensão da classe “*Generic Servlet*”) para receber os parâmetros de entrada e dados do problema “Job Shop”. O *Servlet* realiza as críticas de entrada e, assim, valida estes dados.

Uma sessão é criada quando o usuário faz uma requisição ao sistema e esta sessão administra as interações de cada usuário durante o acesso ao sistema. São também capturadas algumas informações de requisição, como o “Internet Protocol” (IP) do usuário: ao identificar o IP é possível validar ou não o acesso deste ao sistema. Deste modo, um *log* é mantido contendo as informações de atividades das pesquisas e testes de todos os usuários para controle do administrador do sistema.

Todos os erros ocasionados durante a sessão do usuário são capturados e redirecionados ao *Servlet*, onde estes erros serão tratados, incluindo os erros gerados pela GPU ou pela placa gráfica que se encontra instalada em outra plataforma de *hardware*, além do *Host*. Este erro é dinamicamente registrado no log do sistema e é prontamente respondido ao navegador (cliente) através de mensagem ao usuário sobre o erro ocorrido.

O recurso de *upload* de arquivos entre o cliente (navegador) e o servidor foi inserido na arquitetura do servidor: é utilizado uma API “*Smart Upload*”, também *software* livre. O pesquisador poderá utilizar esta ferramenta para enviar um arquivo contendo a descrição de um problema “Job Shop”. Este arquivo é formatado conforme os existentes na *OR-Library* e será descrito em mais detalhes na Seção 5.8.

Todas as informações do problema “Job Shop” processadas pelo sistema são enviadas ao navegador (cliente) e encapsuladas em um documento PDF, possibilitando ao usuário (pesquisador) a flexibilidade de uso dos diversos recursos desta formatação de documento. Este formato é um padrão internacional para compartilhar documentos eletrônicos, e, além de visualizar, o usuário poderá pesquisar, verificar, imprimir e compartilhar arquivos.

A geração do documento PDF é feita dinamicamente através de uma biblioteca *iTex*²³ (versão 2.1.5). Este é um *software* livre que pode modificar, adicionar marcas-d'água e assinar digitalmente o documento (não implementado no Simulador), proteger, mesclar múltiplos documentos e outras funcionalidades.

Um relatório de *Gantt* (Seção 2.6) é gerado a partir do *software* livre *JfreeChart* (versão 1.0.13) que é uma biblioteca totalmente escrita em *Java* e pode ser utilizado para gerar diversos gráficos, como o de *Pizza*, *Barra*, *XY*, *Linha* e outros. Além de utilizar o Gráfico de *Gantt*, o Simulador fornece um gráfico *XY* ao pesquisador podendo este acompanhar a evolução dos valores de *makespan* encontrados a cada iteração. Uma característica importante desta biblioteca é que ela é totalmente integrada ao *iText*.

Abaixo, é descrito como gerar este documento. Em resposta à requisição do cliente (navegador) o *Servlet* realiza os seguintes passos:

- Obtém uma referência de *OutputStream* para retornar uma saída ao cliente;
- Um *content type* é selecionado para *Application/Pdf* para informar o cliente (navegador) que um relatório no formato PDF será enviado;
- Criar o relatório passando para ele os dados que geraram o gráfico e a imagem do gráfico gerada.
- Cria os gráficos;
- Fechar o *OutputStream*.

²³ *iText* - <http://itextpdf.com>

5.4.4 Classes Java Beans

Um *Bean* é uma classe Java obedecendo a um conjunto de regras simples de *design* e nomeação seguindo a especificação *Java Beans*²⁴.

Java Beans são componentes de *software* para construção de aplicações e apresenta a vantagem de ser reutilizável e distribuído, constituindo uma ferramenta para manipulação de dados. Estes componentes apresentam um conjunto bem definido de propriedades e métodos, permitindo que estas propriedades sejam alteradas e métodos sejam invocados.

Métodos foram construídos para o Simulador de modo a representar todas as informações de uma instância do problema *Job Shop*. Por exemplo, as informações de número de tarefas, número de robôs, e XML são obtidas através de métodos *getTarefas()*, *getRobos()* e *getXML()*, respectivamente. Estas informações estruturadas são convertidas em mensagens e enviadas para a GPU, ou seja, métodos *Java Beans* organizam e armazenam as informações da *instância* do problema de escalonamento de forma a manipular e reutilizar as informações nos documentos gerados pelo Simulador no formato PDF.

Com o uso da interface *Java Native Access*²⁵ (JNA), descrita nas próximas páginas, todos os parâmetros do problema *Job Shop* são enviados para o ambiente da GPU através de uma meta linguagem XML construída pelo servidor de aplicação, a qual é escrita na memória do referido servidor para uma biblioteca dinâmica “libjnaxxx.so”. Esta biblioteca dinâmica consiste em um módulo executável invocado pela *interface* JNA e controla a aplicação que executará o algoritmo no dispositivo GPU, que foi construída em outra arquitetura de *software* e *hardware*.

5.5 Mensagens

As mensagens trocadas entre o dispositivo e o servidor de aplicação são feitas por uma meta linguagem XML (“*eXtensible Markup Language*”), que é um formato padrão para a troca e manipulação de dados entre sistemas e plataformas diferentes, assim como pode ser empregado para o armazenamento de informações.

²⁴ **JavaBeans** -<http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html>

²⁵ **JNA** -<https://jna.dev.java.net/>

Existem dois padrões de APIs (“*Application Programming Interface*”) que processam as informações inseridas em uma estrutura de dados XML e disponibilizam estas informações para a aplicação: o primeiro padrão é baseado em objetos ou DOM (“*Document Object Model*”) e o segundo padrão é baseado em eventos ou SAX (“*Simple API for XML*”).

A implementação feita em DOM apresenta a desvantagem de utilizar muita memória para manter a estrutura de dados em uma árvore binária, para o armazenamento e representação do documento. Esta representação em memória é maior do que o armazenamento do arquivo texto em disco, devido ao uso de ponteiros na representação das associações entre os diversos elementos do documento, por exemplo, um único nó na árvore é representado através de ponteiros para o pai do nó, os filhos do nó, seus parentes e outros.

5.5.1 API SAX

O dispositivo Simulador apresentado nesta dissertação utiliza o segundo padrão de API para o processamento das informações: o padrão de API SAX é a escolha mais adequada ao dispositivo Simulador devido às necessidades de menor consumo de memória deste dispositivo Simulador e apresenta melhor desempenho na transferência das informações entre a arquitetura do dispositivo GPU e o *Host*.

Para a API SAX é disponibilizado um analisador (*parser*) *Java*, o qual possui a característica de portabilidade e independência entre as duas plataformas. Isso facilita o desenvolvimento de novas aplicações de algoritmos paralelos para o Simulador. Desta forma, as trocas de mensagens entre o dispositivo GPU, escrito em linguagem C, e o servidor de aplicação, escrito em *Java*, são feitas no formato de documento XML.

O Simulador utiliza a API *gnujaxp*²⁶ da “*Free Software Foundation*” por ser uma implementação *livre* do padrão XML disponibilizada em API para *Java*. Esta biblioteca é instanciada e utilizada pela classe *Java Beans* descrita na Seção 5.4.4.

As variáveis de entrada do problema “*Job Shop*” são limitadas pela própria definição do problema, por exemplo: informações de números de tarefas (*jobs*); números de

²⁶ *gnujaxp* - <http://www.gnu.org/software/classpathx/jaxp/>

operações por tarefa; número de robôs (máquinas) e seus tempos ou pesos e sequências de tarefas, todas de pequena ordem de grandeza.

5.5.2 Acessando Bibliotecas Nativas

Todo o ambiente do servidor de aplicação utiliza tecnologia *Java* que compõem os métodos necessários para que o Simulador seja disponibilizado na *web*. No entanto, este Simulador utiliza técnicas de processamento paralelo existente em uma placa gráfica para resolver problemas computacionais complexos do tipo “*Job Shop*”, em uma fração do tempo necessário.

A tecnologia CUDA, descrita no Capítulo 3, é gratuita e de fácil implementação. A NVIDIA adotou C como padrão de linguagem para processar algoritmos no dispositivo GPU. Esta linguagem é uma das mais utilizadas para programação de baixo nível e é a única, atualmente, suportada pela empresa a qual permite aos desenvolvedores escreverem *software* paralelo, recorrendo ao processamento de múltiplos núcleos existentes na GPU.

Esforços foram feitos para que outras linguagens de ligação (“*binding*”) fossem integrados à tecnologia CUDA, como PyCuda²⁷ (*Python*) e Jcuda²⁸ (*Java*). Entretanto, estas linguagens não acessam todas as funcionalidades disponíveis na arquitetura CUDA e não foram extensivamente testadas. A NVIDIA, conforme já dito, não fornece suporte para estas linguagens.

O Simulador utiliza a linguagem C para a execução do algoritmo *Busca Tabu* de modo a obter-se robustez nesta solução. Neste contexto, é necessário fazer uso de técnicas aprimoradas para executar os algoritmos escritos na linguagem C, invocados a partir do servidor de aplicação (ambiente *Java*). Uma solução prática é utilizar o recurso em bibliotecas dinâmicas alocadas no sistema operacional.

Este acesso é possibilitado através de chamadas a métodos nativos ou “*Java Native Access*”²⁹ (JNA). Esta interface disponibiliza um acesso às bibliotecas compartilhadas no sistema operacional (so no *Linux* ou *dll* no *Windows*) utilizando apenas código *Java*. O acesso é dinamicamente realizado em tempo de execução.

²⁷ PyCuda - <http://mathematixian.de/software/pycuda>

²⁸ Jcuda.org – Jcuda - <http://www.jcuda.org/jcuda/JCuda.html>

²⁹ JNA - <https://jna.dev.java.net/>

Portanto, o JNA permite chamar diretamente a função nativa escrita na linguagem C utilizando métodos de invocação *Java*. As chamadas não requerem manipulações especiais ou configuração.

A classe *GerarMakespan (Java) do projeto* é responsável pela chamada das funções nativas, escritas em C, alocadas em uma biblioteca dinâmica. Utilizando o método *acionarCuda (Java)* e sua respectiva interface (*Java*) *AcoesEmCuda*. Este processo ocorre em *Native.loadLibrary* (“nome da biblioteca”). Este nome começa com “lib...” e possui a extensão “.so” (em ambiente Linux) e maiores detalhes sobre objeto compartilhado serão descritos na Seção 5.5.4.

Principais Características:

- É executado em várias plataformas que suportam *Java*;
- Mapeamento automático feito no ambiente *Java* para funções nativas;
- Conversão automática entre Strings de C e *Java*;
- Argumentos em Estruturas e Uniões;
- Apontadores de funções, (chamadas de retorno de códigos nativos para o *Java*) como argumentos e/ou membros de uma estrutura (*struct*);
- Diretórios-padrão onde se deve colocar a biblioteca compartilhada, entre eles o /usr/lib, que simplesmente exige a criação de um link simbólico para o arquivo real.

5.5.3 Tipos de Dados Compartilhados

Utilizando o JNA, os tipos primitivos em *Java* (objetos equivalentes) mapeiam diretamente os tipos nativos em C de mesmo tamanho. Tipos sem sinal (*unsigned*) utilizam o mesmo mapeamento dos tipos com sinal (*signed*).

Tabela 7: Tipos de dados primitivos equivalentes entre C e Java.

Tipo Nativo em C	Tamanho	Tipo Java
Char	8-bit integer	byte
Short	16-bit integer	short
wchar_t	16/32-bit integer	char
Int	32-bit integer	int

Int	boolean value	Boolean
Long	32/64-bit integer	NativeLong
long long	64-bit integer	Long
Float	32 bit FP	Float
Double	64-bit FP	Double
char*	C string	String
void*	pointer	Pointer

Por exemplo: os valores de números de robôs, números de tarefas e de *makespan*, são informações do tipo “*unsigned short integer*”, o arquivo XML é do tipo *char** em C e *String* no Java.

5.5.4 Objeto Compartilhado

Objeto compartilhado, “*Shared Object*” (.so) ou “*Shared Library*”, é uma biblioteca compartilhada existente no sistema operacional Linux, e é similar às bibliotecas dinâmicas (.dll) encontrado no sistema operacional Windows.

Por padrão, Java segue as convenções do sistema de acolhimento para o carregamento de bibliotecas nativas, ou seja:

- No Windows, as bibliotecas serão pesquisadas na PATH.
- No Unix/Linux, elas são procuradas no LD_LIBRARY_PATH.
- No Mac OS, a variável de ambiente DYLD_LIBRARY_PATH é utilizada.

Também é possível passá-la, alterando a variável de sistema *Java.library.path* (*jna.library.path*) para JNA. Se a biblioteca não faz parte de um dos diretórios especificados a execução de programa irá gerar um *UnsatisfiedLinkError* ...

Do mesmo modo, cada sistema tem as suas próprias convenções para nomear arquivos representando bibliotecas.

- Em Unix / Linux, utiliza-se o prefixo *lib* juntamente com a extensão “.so”. Assim seja “libjnaxxx.so”.

- No Mac OS, pode ser utilizado o prefixo *lib* juntamente com a extensão. “.jnilib” ou *libjnaxxx.jnilib*.

No presente trabalho utilizamos o SO Linux e os métodos desenvolvidos, após sua compilação, são armazenados em bibliotecas com nome começando por “**libjna**” e terminando com a extensão “.so”. Estes ficam localizados no diretório **/lib** do sistema operacional Linux ou pode-se utilizar um *link* apontando para a real localização do arquivo. Por exemplo, no Simulador o objeto gerado está localizado no diretório:

/home/helmar/NetBeansProjects/c/jna/jnaajbs/dist/Debug/GNU-Linux-x86.

5.6 Ambiente Integrado de Desenvolvimento

Para o desenvolvimento da arquitetura de *software* utilizou-se o Netbeans³⁰ (versão 6.8) como um ambiente de desenvolvimento integrado ou “*Integrated Development Environment*” (IDE) para a construção das camadas de *softwares* necessárias para o desenvolvimento do Simulador aqui proposto.

O IDE é “*open source*” e totalmente escrito em Java. Além das linguagens de programação C e Java utilizados no Simulador, este ambiente IDE suporta outras linguagens como PHP, Fortran, Groovy e Ruby.

Com o suporte às plataformas de *software* C e Java, pode-se integrar estas camadas de *softwares* em projetos definidos pelo IDE, estruturando e organizando todo o trabalho de desenvolvimento. Uma grande vantagem é a portabilidade deste ambiente de desenvolvimento para qualquer plataforma que suporte Java, C e CUDA.

O IDE auxilia a programação do Simulador, sua compilação, o *debug* e instalação da aplicação e este é integrado ao servidor de aplicação *Tomcat*. Sua estrutura simplifica o desenvolvimento de *software* e aumenta a produtividade, ideal para uso em um laboratório de pesquisas. Este ambiente único possui todas as funcionalidades necessárias para o desenvolvimento de novos métodos, é amplamente documentado e disponível em português.

³⁰ Netbeans - <http://netbeans.org/>

A NVIDIA disponibiliza suporte ao CUDA para as plataformas Linux, Windows e Mac. É fornecido um *Toolkit* de instalação, um driver dedicado para executar a aplicação na GPU e manuais do sistema.

Um pré-requisito ao projeto do Simulador foi considerar a utilização de apenas *softwares* livres ou gratuitos de modo a termos baixo custo no ambiente de desenvolvimento. Neste contexto, escolheu-se utilizar o Linux Ubuntu (versão 8.04) como sistema operacional. Instalou-se a ferramenta de desenvolvimento para a plataforma Java, JSE versão 1.6.0_11. Todos os *softwares* são obtidos através de *downloads* feitos na *internet* diretamente dos sites de cada organização.

O Simulador utiliza o *driver* da NVIDIA para Linux (32 bits), versão 185.18.14, utilizado para controlar o processador gráfico GeForce 9800GT, este com 512 MB (DDR3) para memória global.

Abaixo, é descrito o ambiente utilizado para o desenvolvimento do Simulador instalado no laboratório de robótica CONTROLAB - NC/UFRJ:

- Versão do gcc 4.2.4;
- Versão do CUDA SDK 2.2;
- Versão do CUDA toolkit 2.2 para Ubuntu 8.04 versão 32 bits;
- GeForce: 9800 GT ³¹;
 - 112 núcleos processadores;
 - Memória: 512 MB DDR3;
 - Clock da GPU 600 MHz;
 - Shader clock 1,5 GHz;
 - Velocidade de Memória: 1.800 MHz;
 - Interface de memória: 256 bits;
- Interface PCI-E 2.0
- Ubuntu 8.04 (hardy) ;
- Kernel Linux 2.6.24-25 generic;
- GNOME 2.22.3;
- Memória 2 GB;
- Processador AMD Athlon™ 64 x 22 Dual Core Processor 4600+;

³¹ GeForce: 9800 GT http://www.xfxforce.com/ecms.ashx/0207834f-1e9e-4feb-9191-332cb75c06cf/GraphicCardSeriesRelated_RelatedGraphicCardModels11/XFX_9800GT_DLP.pdf

- Espaço disponível em disco: 131 GB.

Como anteriormente informado, o IDE Netbeans é amplamente utilizado por desenvolvedores e é flexível de modo colaborativo para o desenvolvimento de projetos entre equipes de laboratório.

5.6.1 Instalação do Netbeans

A seguir serão descritas as etapas de procedimento a serem realizadas para a instalação deste ambiente gráfico de desenvolvimento:

- Realizar o *download* e instalar o kit de desenvolvimento *java* (JDK) versão 6 ou superior. Pode ser utilizado o gerenciador de pacotes *Synaptic*, disponível no sistema operacional linux Ubuntu;
- Instalar o Netbeans 6.8 ou versão superior;
- Alternativamente pode-se fazer o download completo da distribuição do java JDK (java SE 6) e do ambiente de desenvolvimento integrado (IDE) no site da Oracle;
- Após a instalação e configuração das permissões e variáveis de ambiente do linux, em `home/conta/.bashrc`, execute a IDE para inicializar a configuração de ambiente;
- Acesse “Opções” na aba “Ferramentas”, acione o botão “Miscelânea” e a aba “Arquivos”. Acione o botão *novo* e insira o texto em “*extensão do arquivo*” **cu**. Acione o botão de “ok”. Associe o tipo de arquivo recém inserido a “Arquivos C (text/x-c)”. Os arquivos compilados pelo *nvcc* da NVIDIA possuem a extensão **cu**;
- Em *Opções* acione o botão “C/C++”, adicione o compilador *nvcc* em “Ferramentas de construção”, seguindo a orientação da Figura 24 e dos locais escolhidos para a instalação da IDE e do projeto.

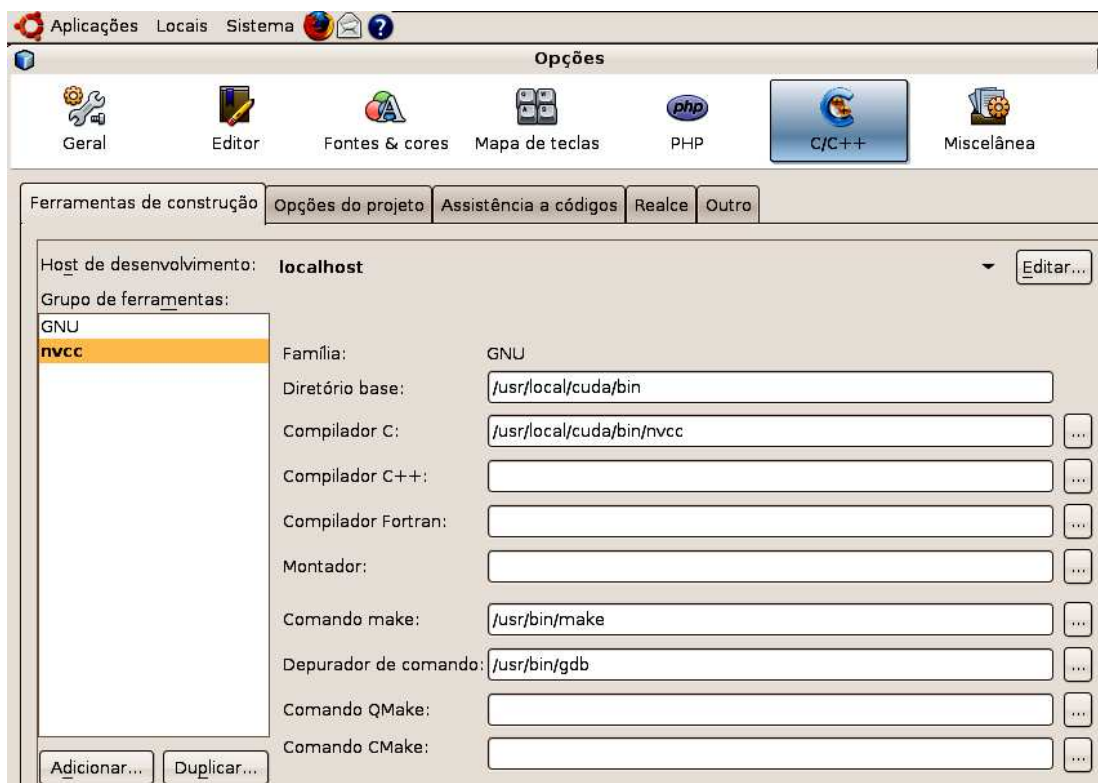


Figura 24: Configuração do compilador *nvcc* na IDE.

- Na aba “Assistência a códigos” siga a orientação da Figura 25 para inserir os diretórios de inclusão e *macros* utilizadas pelo *nvcc*.

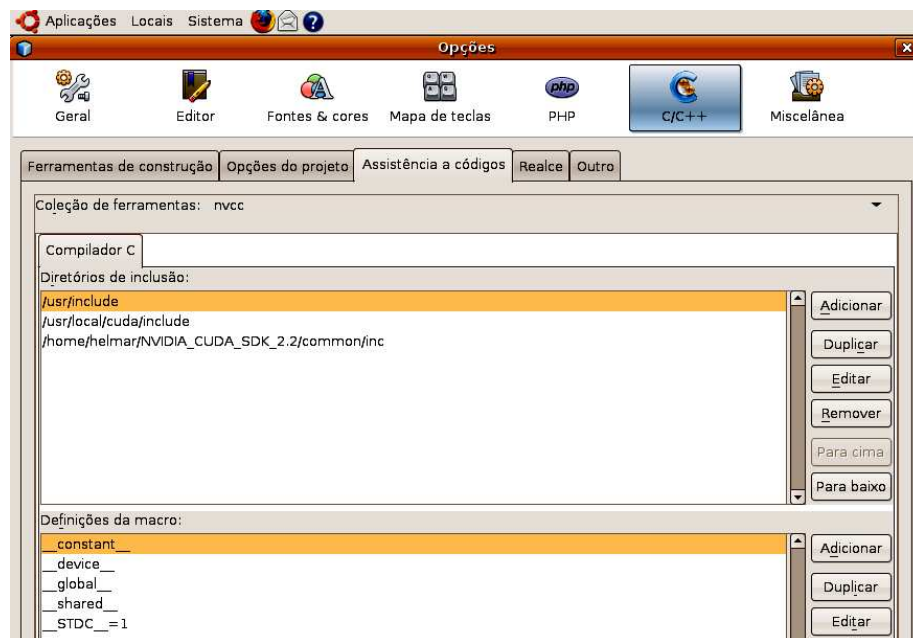


Figura 25: Configuração dos diretórios de inclusão e as macros utilizadas no projeto.

5.6.2 Abrindo um novo projeto

Para abrir um novo projeto de forma a desenvolver e adicionar um novo algoritmo ao Simulador é fácil e rápido, utilizando-se apenas o ambiente do NetBeans (IDE) e com o seu *wizard*. Este apresentará passo a passo os procedimentos necessários para a abertura do novo projeto, como a seguir:

- Abra a aba “Arquivo”, escolha a opção “Novo Projeto”;
- Escolha o projeto e sua categoria C/C++, projetos, projetos “Biblioteca estática de C/C++” e acione o botão de próximo;
- Defina um nome para o projeto e a sua respectiva localização o qual deverá ser a mesma dos outros projetos (“Netbeans Projects”) já construídos;
- Passe o mouse no projeto recém aberto e com o botão direito acesse “Propriedades” do projeto para a configuração deste ambiente;
- Em “Construir”, escolha a “Coleção de ferramentas” o compilador *nvcc*.

5.6.3 Projetos para o Simulador

Três projetos foram abertos nesta IDE, um para cada plataforma de *software*, com finalidades distintas no funcionamento do Simulador, são eles:

- *webaajbs* - aplicação *web* escrita em java (compilador *java*) implementando uma arquitetura MVC em um servidor de aplicação;
- *jnatsjbs* – aplicação C (compilador *gcc*) desenvolvida para ser a interface entre as duas arquiteturas: servidor de aplicação *web* e a plataforma CUDA.
- *tsjbs* – aplicação escrita em C (compilador *nvcc* - NVIDIA) com a arquitetura CUDA utilizando os recursos da computação paralela disponíveis na GPU e executando o algoritmo *Busca Tabu*.

As camadas de software construídas para o Simulador podem ser vistas em detalhes nas Figuras 27, 28 e 29.

Ao acessar a IDE visualizam-se os projetos através da aba de *Projetos* conforme mostrado na Figura 26. Esta facilidade auxilia os desenvolvedores na criação de aplicativos híbridos em diferentes plataformas.



Figura 26: Três projetos do simulador criados no ambiente da IDE e detalhe ao lado do projeto “tsjbs”.

Cada projeto pode ser aberto pelo desenvolvedor. Sua estrutura é detalhada de modo a possibilitar um rápido acesso ao módulo que se deseja construir ou alterar, acessando o projeto escolhido na pasta de projetos. Utilize o botão direito do mouse para acessar as funcionalidades da IDE para:

- Compilar (Limpar e Construir);
- Renomear projeto;
- Copiar;
- Excluir;
- Adicionar novos arquivos fontes classe c ou h (header);
- Acessar as propriedades do projeto.

Por exemplo, pode-se abrir o programa (Figura 26, lado direito) fonte (kernelJBS.cu) que é o *kernel* do projeto *tsjbs*, para programação dando um duplo clique no nome do arquivo. Podemos compilar este utilizando “Limpar e Construir”.

Este módulo contém o algoritmo paralelo a ser executado exclusivamente no *dispositivo* GPU e executa o algoritmo paralelo *Busca Tabu* descrito no Capítulo 3. O desenvolvedor pode assim, realizar as alterações ou correções que se fizerem necessárias através de um editor de texto da própria IDE.

5.6.4 Ambiente Web e Java

Para facilitar a compreensão de como se processa a iteração entre os ambientes de *hardware* e seus relacionamentos (Figura 27) será descrito abaixo a interação do

cliente com o servidor oferecendo as funcionalidades do Simulador do problema “*Job Shop*”:

- Um navegador (cliente) solicita uma requisição ao servidor de aplicação, via servidor *web* (Tomcat). Por exemplo, processar uma instância definida do problema “*Job Shop*”;
- O servidor de aplicação organiza e estrutura as informações do problema, confirma sua integridade e as envia para sua interface JNA em mensagens XML;
- O executável, na forma de biblioteca dinâmica do sistema operacional, recebe estas mensagens e reestrutura estas na forma de estrutura de dados e parâmetros de funções em C;
- Esta estrutura de dados do problema é encaminhada para o *kernel* do dispositivo GPU onde o algoritmo *Busca Tabu* é processado nas centenas de núcleos;
- Uma solução ao problema é encontrada pelo algoritmo e esta é armazenada em outra estrutura de dados (*struct*);
- As informações são recebidas pela interface em C e codificadas em padrões XML para envio ao servidor de aplicação via interface JNA;
- O servidor de aplicação recebe a solução, gera o gráfico de Gantt e insere estes objetos e informações do problema na estrutura do documento PDF;
- Uma resposta é encaminhada ao navegador (cliente) pelo servidor de aplicação com o documento PDF e o navegador, através de seu *plugin*, abre o dito documento. Este torna-se disponível ao usuário para estudos e avaliações do problema com o método utilizado.

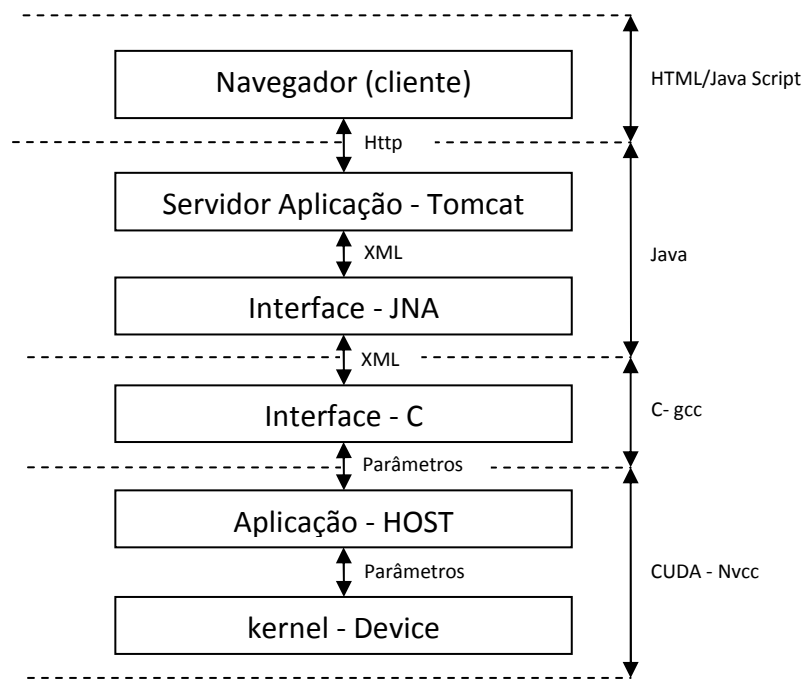


Figura 27: Arquitetura em camadas de *software* do simulador.

5.6.5 Ambiente C – Interface JNA

Através da Figura 28 pode-se visualizar os módulos deste ambiente. A interface C manipula as informações do problema JSP, modela as estruturas de dados utilizando o tipo *struct* da linguagem C e modela a solução do problema codificando-a em um arquivo XML, escrito em memória, como a seguir:

- *jnatsjbs* – interface C entre as arquiteturas de software. É invocada pela interface Java e realiza as transferências de informações entre os dois ambientes. Esta troca de informações se dá através de métodos do tipo *get* e *set*;
- *modelarJob* – modela as informações do problema (dados de entrada) em uma estrutura de dados do tipo *struct*;
- *modelarSol* – modela as informações de saída com a solução do problema, no formato XML;
- *substring* – função que retorna uma *substring* de uma *string*;

O projeto *jnatsjbs* utiliza a biblioteca estática ***libtsjbs.a*** construída pelo projeto *tsjbs*, o qual possui as funções kernel para o algoritmo paralelo *Busca Tabu*, as bibliotecas ***libcudart.so*** e ***libcutil.a*** utilizadas pelo CUDA da NVIDIA.

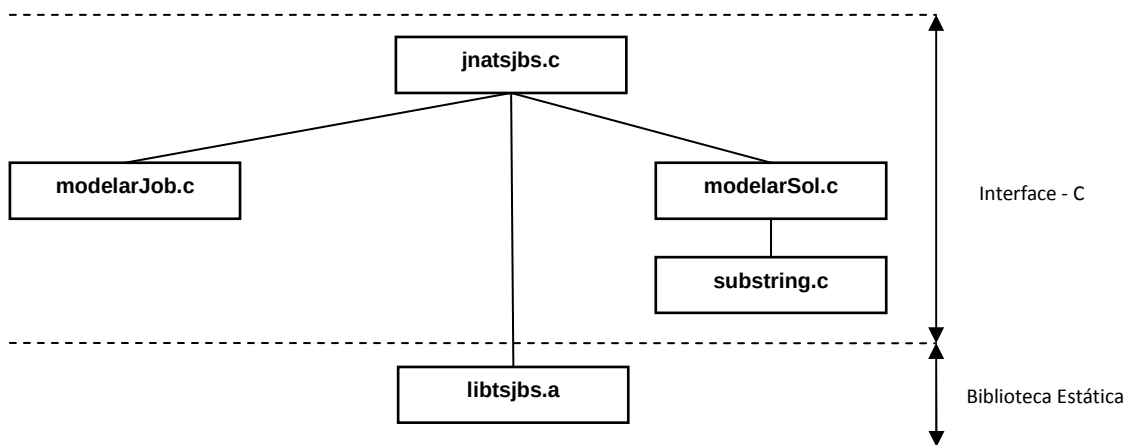


Figura 28: Módulos do *software* que compõem a *interface C*.

5.6.6 Ambiente CUDA

O ambiente CUDA, descrito na Seção 4.11, foi inserido na arquitetura de *software* do Simulador (Figura 27) o que permite o uso da computação paralela disponível no dispositivo GPU.

O módulo *kernel* é separado dos demais pelo compilador da NVIDIA, o *nvcc*, através de notação “<<< >>>” inserida no código fonte (Seção 4.11.4) do módulo principal: o *kernel* é invocado pelo módulo *tsjbs* (Aplicação – *Host*) no instante da compilação para gerar o objeto executável que processará o algoritmo *Busca Tabu* no ambiente paralelo. Como devidamente configurado, é gerada uma biblioteca estática *libajbs.a* onde as suas funções poderão ser invocadas pela interface descrita anteriormente.

- *tsjbs* – programa a ser executado no *Host*, manipula a estrutura de dados do problema JSP (dados de entrada e de saída) enviando-a ao dispositivo GPU, no ato de execução e recebe a estrutura de dados com a solução definida pelo algoritmo *tabu paralelo*, sendo esta executada no *kernel* do dispositivo.
- *impTar* – módulo que mostra, em um console, de forma textual os dados da estrutura de entrada.
- *iniSol* – inicializa a estrutura de dados que conterá uma solução ao problema JSP.
- *impSol* – semelhante ao módulo *impTar*, este módulo mostra, em um console, também de forma textual, os dados da estrutura de dados que contem a solução encontrada ao problema JSP. Este módulo é muito útil para a

depuração e desenvolvimento do algoritmo que define uma solução ao problema JSP.

- *integerToString* – converte um valor do tipo *integer* para uma *String* ou cadeia de caracteres.
- *inrand* – gera números aleatórios para uso do algoritmo existente no *kernel*.

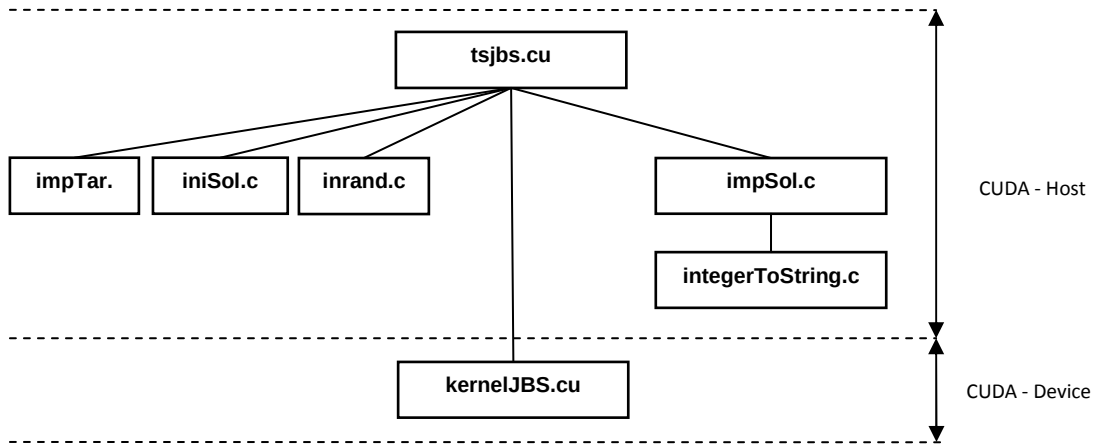


Figura 29: Módulos do *software* que controla o *dispositivo GPU*.

5.6.7 Escrever um novo Método utilizando *Templates*

O ambiente construído para o Simulador, descrito na presente dissertação, permite ao pesquisador o uso de *template* que pode ser herdada do *kernel* do Simulador *busca tabu*, facilitando a construção de novos métodos paralelos. Conjuntamente com o uso da IDE, este ambiente do Simulador permite um desenvolvimento rápido ao pesquisador e este ficaria apenas focado no desenvolvimento do novo método de seu interesse, não se preocupando com as interfaces, interações com o usuário do sistema, estruturas de arquivos, consistências de dados armazenamento de soluções e etc.

Abaixo são descritas as funcionalidades comuns a diversos métodos que poderiam ser utilizados pelo pesquisador:

- Estruturas de arquivos descritas no Capítulo 3;
- Função de troca (swap) entre elementos de uma lista;
- Função *max()* determinando o maior valor entre dois números inteiros;
- Função de ordenação em uma lista;
- Função de movimento entre operações pertencentes a um caminho crítico;
- Função de cálculo dos tempos de escalonamento de uma solução ou *Scheduling*;

- Função de construção de solução inicial viável;
- Função de identificação do caminho crítico;
- Geração de números aleatórios para o *kernel*.

5.7 Funcionalidades

Nesta seção, serão apresentadas as interfaces do usuário que acessa o Simulador através do seu navegador conectado a uma rede intranet. Inicialmente o usuário deverá acessar a página principal do sistema utilizando o endereço URI ("*Uniform Resource Locator*") do laboratório através de qualquer navegador disponível no mercado.



UFRJ - Universidade Federal do Rio de Janeiro



Dissertação: Simulador "web" de algoritmos paralelos para problemas de escalonamento "Job Shop".

Discente: [Helmar Alvares](#)

Orientadores: Ph.D. Eliana P. L. Aude e Ph.D. Ernesto Prado Lopes

Número de Tarefas

Informe o número de Tarefas a simular: $j = \{0, 1, 2, \dots, 99\}$

Número de Robôs

Informe o número de Robôs a simular: $r = \{0, 1, 2, \dots, 99\}$

- A tabela abaixo mostra o nome de instâncias de problemas, as suas dimensões (número de tarefas - jobs - e de Robôs - máquinas), e a melhor solução conhecida ("Lower Bound"). Estes conjuntos de dados foram desenvolvidos para testes dos problemas JSP e estes foram utilizados como o benchmark nos estudos do Simulador.

- Os problemas "OR-Library" foram descritos originalmente por J.E. Beasley, em "OR-Library: distributing test problems by electronic mail", *Journal of the Operational Research Society* 41(11) (1990) pp1069-1072 e podem ser encontrados em [Welcome to OR-Library](#)

- Faça o upload de um Problema [aqui](#)

Problema	Instância	Lower Bound
CH	Universität Bern 5 x 5 - 5 tarefas x 5 robôs x 5 operações	17
MT06	Muth & Thompson 6 x 6 - 6 tarefas x 6 robôs x 6 operações	55
MT10	Muth & Thompson 10 x 10 problem via S. Lawrence - 10 tarefas x 10 robôs x 10 operações	930
ABZ5	Balas 10x10 problem via S. Lawrence - 10 tarefas x 10 robôs x 10 operações	1234
ABZ7	Balas 15 x 20 problem - 20 tarefas x 15 robôs x 15 operações	665
SWV11	Storer, Wu, and Vaccari hard 50x10 instance - 50 tarefas x 10 robôs x 10 operações	2983

Figura 30: Página inicial do Simulador *web*.

Nesta página principal o usuário encontrará um conjunto de problemas exemplos (descrito no *frame* inferior da página) e poderá realizar *upload* para mais de 60 problemas padrões ("*benchmark*"), encontrados na literatura ou descrever um novo

problema, preenchendo as informações nas caixas de diálogo “Número de Tarefas” e “Número de Robôs”.

5.7.1 Escrever uma nova Instância

1. Informar o número de tarefas e número de robôs, acionar o botão “Informar Operações”.

The screenshot shows two input fields. The first is labeled 'Número de Tarefas' and contains the value '3'. To its right is the text 'Informe o número de Tarefas a simular: j = {0,1,2...,99}'. The second is labeled 'Número de Robôs' and also contains the value '3'. To its right is the text 'Informe o número de Robôs a simular: r = {0,1,2...,99}'. Below these fields is a button labeled 'Informar Operações'.

Figura 31: Detalhes das caixas de diálogo para dados de entrada, número de *job* e número de Robôs.

2. Informar os dados, manualmente, do problema JSP, com as operações a serem escalonadas. O sistema apresenta uma seqüência de páginas dinâmicas (“Java Server Pages”) organizando intuitivamente a estrutura de dados do problema formulado:

Descrição da Tarefa 00:

The screenshot shows a single input field containing the value '3'. To its right is the text: "Informe o número de operações da Tarefa 00 a simular: o = {0,1,2...,99} e tecle ENTER para continuar."

Figura 32: Detalhes das caixas de diálogo para dados de entrada, descrição da primeira tarefa.

3. As informações registradas para o número de tarefas e número de máquinas ou robôs vão induzindo o usuário a preencher corretamente a estrutura de dados a serem utilizados como entrada do problema.

The screenshot shows a sequence of input fields and dropdown menus. It starts with 'Início' followed by an arrow, then 'O00', a text box with '3', a dropdown menu showing 'Robô0', an arrow, 'O01', a text box with '5', a dropdown menu showing 'Robô1', an arrow, 'O02', a text box with '2', and a dropdown menu showing 'Robô?'. The dropdown menu is open, showing options: 'Robô?', 'Robô0', 'Robô1', and 'Robô2'. Below the sequence is a 'voltar' button with a circular arrow icon. The sequence ends with an arrow pointing to 'Fim'. Above the input fields is a header text: 'Tarefa 00: Informe para cada operação de uma tarefa os pesos associados na caixa de diálogo e qual o Robô designado para a execução da referida tarefa.'

Figura 33: Detalhes das caixas de diálogo para dados de entrada, descrição dos pesos.

4. A cada iteração, o sistema irá retornando as informações já fornecidas ao usuário para validação. Caso o usuário verifique que a informação foi digitada com erro, este poderá retornar à página anterior e corrigir esta informação.

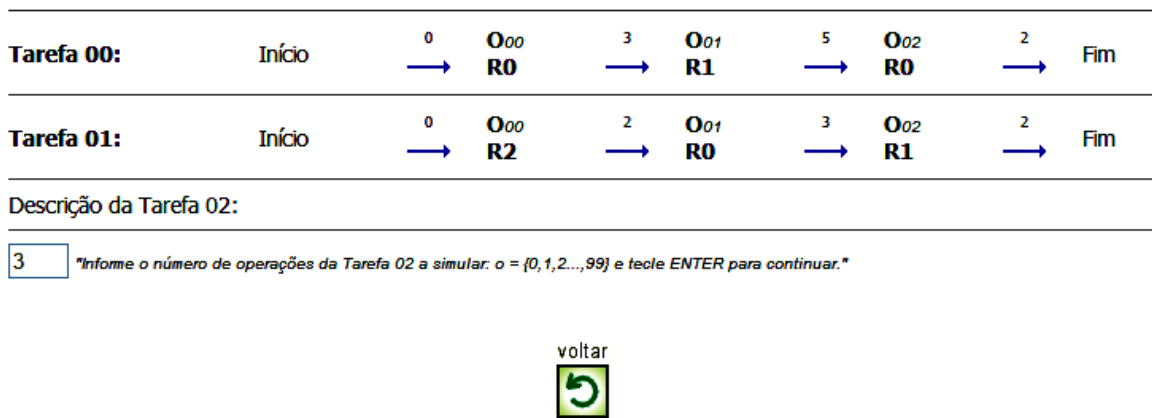


Figura 34: Detalhes das caixas de diálogo para dados de entrada.

5. Ao final da sequência de páginas dinâmicas, o usuário deverá informar:
- Seu e-mail, o qual identifica o usuário do sistema e as soluções encontradas pelo Simulador serão armazenadas em arquivos com a identificação do usuário;
 - Um nome para identificação e título do problema;
 - Qual o método de seu interesse, escolhendo-o na biblioteca de soluções JSP, disponível;
 - O número de Blocos a ser utilizado para processar o problema. O número de Blocos informa quanto de recurso computacional será necessário para executar o algoritmo inserido no método selecionado;
 - O número de evoluções ou iterações que o algoritmo irá processar o método (entre 1 a 999).

Opcionalmente o usuário poderá identificar qual a *thread* que fará a troca entre operações escalonadas na solução corrente, para encontrar o melhor vizinho e auxiliar o aperfeiçoamento do algoritmo em desenvolvimento com o auxílio do Simulador.



Tarefa 00:	Início	0	→	O00 R0	3	→	O01 R1	5	→	O02 R0	2	→	Fim
Tarefa 01:	Início	0	→	O00 R2	2	→	O01 R0	3	→	O02 R1	2	→	Fim
Tarefa 02:	Início	0	→	O00 R2	5	→	O01 R0	6	→	O02 R1	3	→	Fim

Email: Informe o seu E-Mail

Instância: Informe a instância do seu problema.

Solução: Escolha um dos métodos desenvolvidos para o problema JSP.

Blocos: Informe o número de Blocos.

Evolução: Informe o número máximo de iterações - Máximo de 999 iterações.

Manual: Informe a identificação da thread para a busca de vizinhança manual.



Figura 35: Página de conferência dos dados do problema e envio.

5.7.2 Enviar uma Instância - *upload*

Outra forma para apresentar um problema a ser resolvido pelo Simulador é utilizar a funcionalidade de *upload* (Figura 36) descrito no frame central da página inicial, ou seja, um arquivo formatado e estruturado poderá ser enviado ao Simulador:

- Os problemas "OR-Library" foram desc
Journal of the Operational Research Soc
- Faça o upload de um Problema [aqui!](#)

Figura 36: Link na página inicial para *upload* de problemas.

- 1 Uma janela semelhante à descrita no item anterior será apresentada para o preenchimento das informações de identificação do problema. Uma janela "windows" (Figura 37) será ativada ao pressionar o botão "Enviar Arquivo..." ou se o usuário pressionar o botão esquerdo do mouse na caixa de diálogo "Instância".

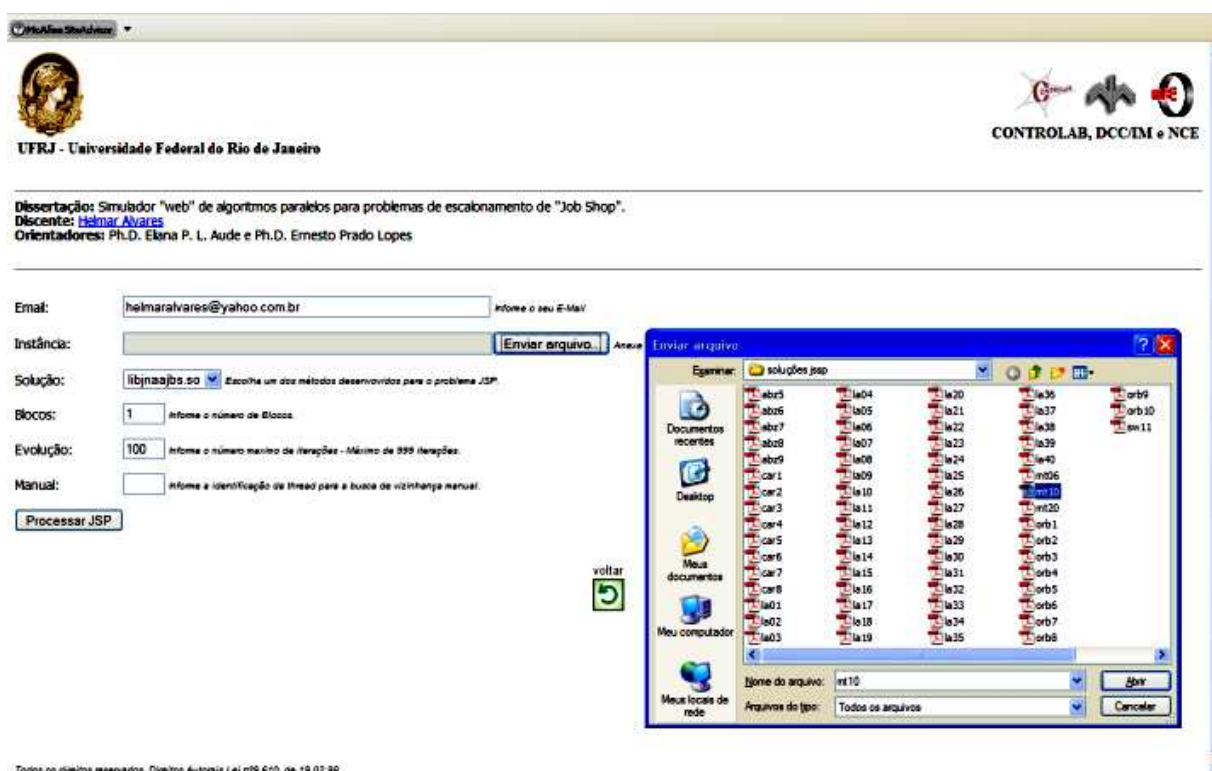


Figura 37: Página de upload de problemas.

Isto é necessário para selecionar o arquivo residente na máquina do usuário que contém os dados já formatados para serem enviados ao Simulador. A estrutura deste arquivo será descrita na Seção 5.8.

5.7.3 Instâncias Clássicas

Finalmente, o usuário poderá escolher um dos problemas exemplos distribuídos pela "OR-Library", para testes e avaliação de algoritmos (Figura 38) para problemas JSP.

Problema	Instância	Lower Bound
MT06	Muth & Thompson 6 x 6 - 6 jobs x 6 robôs x 6 operações	55
MT10	Muth & Thompson 10 x 10 problem via S. Lawrence - 10 operações x 10 jobs x 10 robôs	930

Figura 38: Página inicial com problemas clássicos, exemplo.

5.7.4 Solução em PDF

Após o processamento do problema JSP, uma página no formato PDF (Figura 39) é gerada dinamicamente ("on the fly") pelo Simulador de maneira independente do aplicativo, do hardware e do sistema operacional utilizado. O sistema automaticamente identifica o tamanho do problema de forma a dimensionar este ao

tamanho do documento: o Simulador aceita desde pequenos problemas como 2 tarefas x 2 máquinas, até os maiores problemas descritos na “OR Libray” que é de 50 tarefas x 10 máquinas ou 500 operações de escalonamento. Os detalhes da Figura 39 podem ser melhor visualizados através das Figuras 40 a 44.

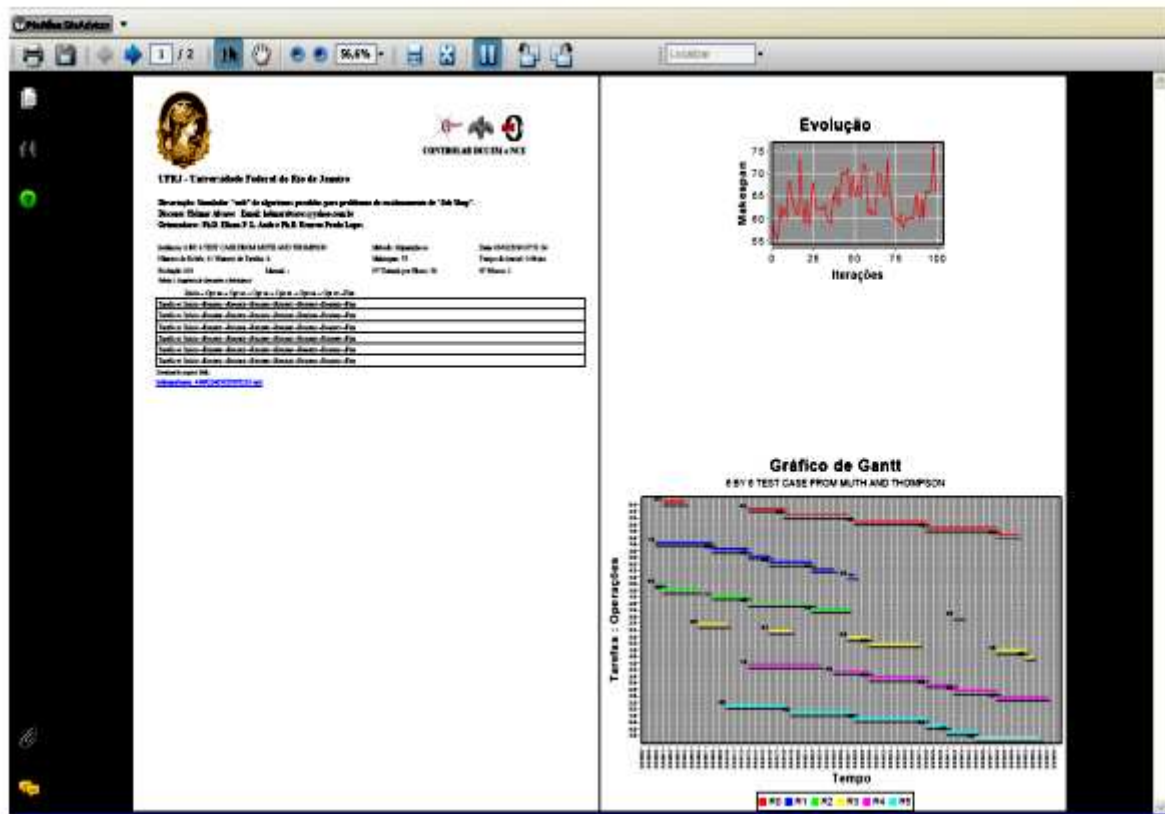


Figura 39: Documento PDF descrevendo uma solução viável.

De acordo com o tamanho do problema, o documento poderá conter uma, duas ou três páginas. O tamanho A4 é o padrão de documento, mas o tamanho A0 poderá ser apresentado para documentar problemas maiores. Este documento poderá ser visualizado pelo próprio navegador, demonstrando todos os detalhes do processamento do problema. Neste documento são visualizados texto, imagens e gráficos, como o *Gráfico de Gantt* e a descrição do problema. Assim, o usuário poderá dar zoom na imagem visualizando detalhes do escalonamento para uma determinada operação, imprimir, salvar em disco ou posta-lo por e-mail:

1. Identificação e descrição do processamento estão logo localizadas no início do documento: identificação da instância ou título do problema; tempo de processamento; número de blocos utilizados pela *grid*; número de *threads*

para cada bloco alocadas para o *Thread Processing Cluster*" (TPC); qual o método da biblioteca escolhido, etc.



UFRJ - Universidade Federal do Rio de Janeiro

Dissertação: Simulador "web" de algoritmos paralelos para problemas de escalonamento de "Job Shop".

Discente: Helmar Alvares Email: helmaralvares@yahoo.com.br

Orientadores: Ph.D. Eliana P. L. Aude e Ph.D. Ernesto Prado Lopes

Instância: Muth & Thompson 10 x 10 problem via S. Lawrence

Número de Robôs: 10 / **Número de Tarefas:** 10

Evolução 5000

Manual: -

Método: libjnatsjbs.so

Início: 1173 **Makespan:** 974

Nº Threads por Bloco: 100

Data: 07/03/2010 15:59:37

Tempo do kernel: 0.053 ms.

Nº Blocos: 1

Figura 40: Exemplo para dados do problema e medidas da solução.

- Logo a seguir é descrito o problema enviado pelo usuário ao Simulador com as informações das tarefas, sequência da alocação das máquinas, operações e tempos de processamento de cada máquina.

Início → Opr 00 → Opr 01 → Opr 02 → Opr 03 → Opr 04 → Opr 05 → Fim
Tarefa 00: Início → R02(001) → R00(003) → R01(006) → R03(007) → R05(003) → R04(006) → Fim
Tarefa 01: Início → R01(008) → R02(005) → R04(010) → R05(010) → R00(010) → R03(004) → Fim
Tarefa 02: Início → R02(005) → R03(004) → R05(008) → R00(009) → R01(001) → R04(007) → Fim
Tarefa 03: Início → R01(005) → R00(005) → R02(005) → R03(003) → R04(008) → R05(009) → Fim
Tarefa 04: Início → R02(009) → R01(003) → R04(005) → R05(004) → R00(003) → R03(001) → Fim
Tarefa 05: Início → R01(003) → R03(003) → R05(009) → R00(010) → R04(004) → R02(001) → Fim

Figura 41: Dados de entrada do problema.

- Link para download do arquivo XML que contém a solução encontrada pelo Simulador. A descrição da estrutura deste arquivo XML e sua definição são vistos através do exemplo abaixo. Esta linguagem de marcação é útil no caso em que o usuário adote a referida solução encontrada e a insira em outro equipamento ou sistema para automação do processo industrial ou comercial.

Download do arquivo XML:

helmaralvares_7600377933645184532.xml

Figura 42: Link ao arquivo XML armazenado no servidor de aplicação.

```

<?xml version="1.0" encoding="UTF-8"?>
<!--UNIVERSIDADE FEDERAL DO RIO DE JANEIRO-->
<!--Discente - Helmar Alvares e-mail: helmaralvares@yahoo.com.br-->
<!--Orientador - Ph.D. Eliana P.L. Aude e Ph.D. Ernesto Prado Lopes-->
<!--robo id - identificação da Máquina ou Robô-->
<!--Op - Operação de uma Tarefa de escalonamento-->
<!--nameOp - identificação (Tarefa:Operação)-->
<!--startValue - início do escalonamento da operação de uma tarefa ou tar-->
<!--stopValue - fim do escalonamento da operação de uma tarefa ou tar-->
<!--Todos os direitos reservados. Direitos Autorais Lei nº9.610, de 19.02.98-->
<!--instancia instancialID="Exemplo 3x3" Jobs="3" Robos="3"-->
<jssp>
  <robo id="R0">
    <op>
      <nameOp>0:0</nameOp>
      <startValue>0</startValue>
      <endValue>3</endValue>
      <weightValue>3</weightValue>
    </op>
    <op>
      <nameOp>2:1</nameOp>
      <startValue>5</startValue>
      <endValue>11</endValue>
      <weightValue>6</weightValue>
    </op>
    <op>
      <nameOp>1:1</nameOp>
      <startValue>11</startValue>
      <endValue>14</endValue>
      <weightValue>3</weightValue>
    </op>
  </robo>
  <robo id="R1">
    <op>
      <nameOp>0:1</nameOp>
      <startValue>3</startValue>
      <endValue>8</endValue>
      <weightValue>5</weightValue>
    </op>
    <op>
      <nameOp>2:2</nameOp>
      <startValue>11</startValue>
      <endValue>14</endValue>
      <weightValue>3</weightValue>
    </op>
    <op>
      <nameOp>1:2</nameOp>
      <startValue>14</startValue>
      <endValue>16</endValue>
      <weightValue>2</weightValue>
    </op>
  </robo>
  <robo id="R2">
    <op>
      <nameOp>2:0</nameOp>
      <startValue>0</startValue>
      <endValue>5</endValue>
      <weightValue>5</weightValue>
    </op>
    <op>
      <nameOp>1:0</nameOp>
      <startValue>5</startValue>
      <endValue>7</endValue>
      <weightValue>2</weightValue>
    </op>
    <op>
      <nameOp>0:2</nameOp>
      <startValue>8</startValue>
      <endValue>10</endValue>
      <weightValue>2</weightValue>
    </op>
  </robo>
</jssp>

```

Exemplo 1: Arquivo XML gerado pelo Simulador, com uma solução viável ao problema MT06 da OR-Library.

4. *Gráfico de linha* ou XY demonstrando a evolução dos resultados de busca local pelo número de iterações definido pelo usuário.

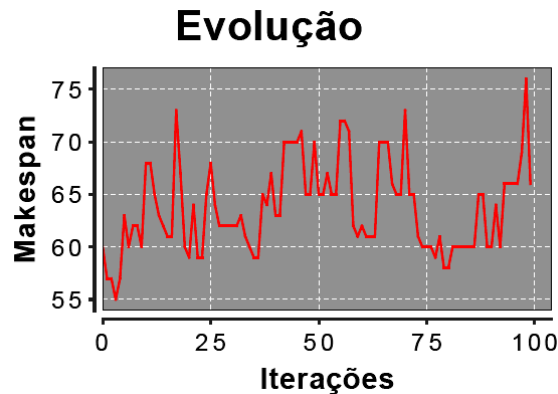


Figura 43: Gráfico de linha demonstrando a evolução por iterações do algoritmo “6 by 6 test case from Muth and Thompson”.

5. *Gráfico de Gantt* com a descrição do título do problema, a ordem das operações para cada máquina ou robô e seu tempo de operação.

Por exemplo, a máquina R0 processa as operações 00 (tarefa 0, operação 0), no tempo inicial 0 e tempo final 3; 11 (tarefa 1, operação 1), no tempo inicial 3 e tempo final 6; 21 (tarefa 2, operação 1), no tempo inicial 6 e tempo final 12.

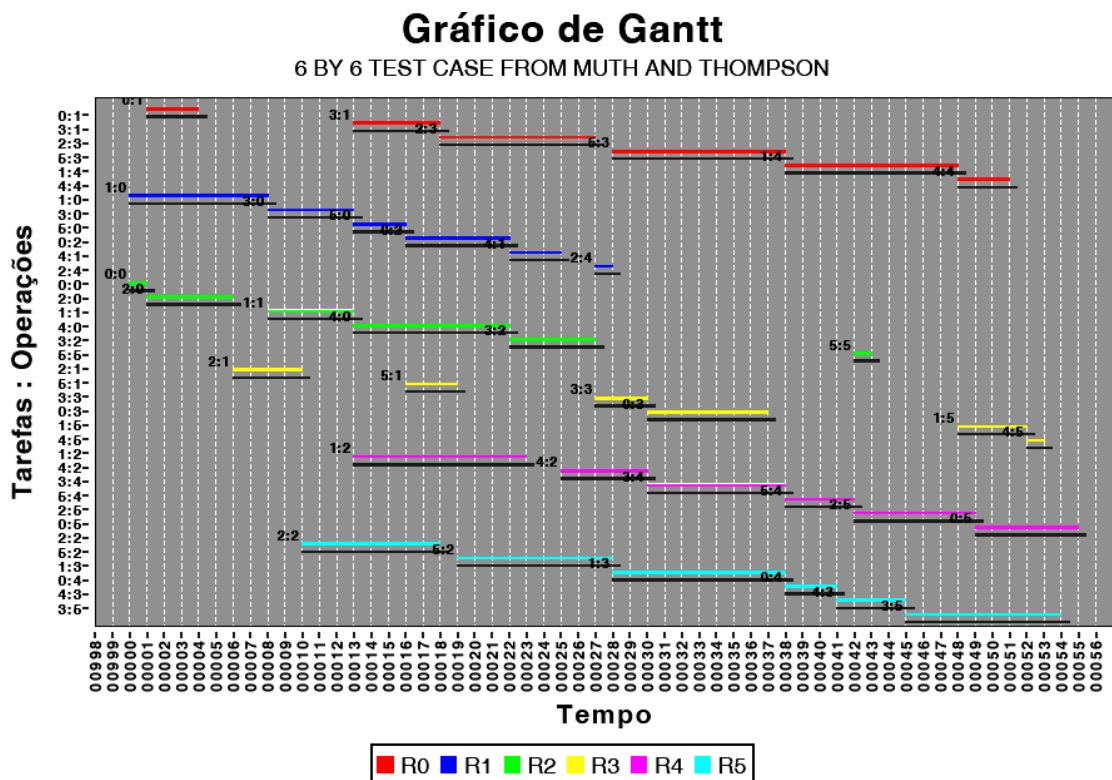


Figura 44: Diagrama de Gantt descrevendo uma solução viável.

5.8 Estrutura do Arquivo de Entrada para *Upload*

O propósito desta seção é o de explicar como os dados do arquivo de entrada estão estruturados e formatados representando, assim, uma instância de problema JSP, a ser simulado no dispositivo apresentado nesta dissertação.

Esta é a mesma estrutura utilizada pela *OR-Library*³², descritos originalmente por J. E. Beasley (1990).

Antes que a execução do escalonamento seja feito, algumas informações básicas do problema serão necessárias, como: o número de operações por tarefa; a ordem para a qual cada tarefa é executada em cada máquina; a quantidade de tempo ou a descrição do peso para cada operação da tarefa despendida para cada máquina.

Por exemplo: no problema 3 x 3, descrito na Seção 2.5, o arquivo apresenta a seguinte formatação:

- As duas primeiras linhas (1 e 2) descrevem o título ou identificação do problema e o número de *tarefas x máquinas*;
- A seguir, cada tarefa é representada por uma linha;
- Cada linha de uma *tarefa* descreve a ordem ou sequência de operações em dupla de números inteiros separados por espaços em branco;
- Cada dupla de par de inteiros representa a identificação da máquina ou robô e consecutivamente, o seu tempo ou peso despendido para cada máquina. Os números inteiros podem ou não ter zeros à esquerda e seus valores variam de 0 a 100 para as máquinas ou robôs e 0 a 100 para os seus tempos ou pesos.

Linha 1 - Exemplo de problema 3 x 3
Linha 2 - 03 03
Linha 3 - 00 03 01 05 02 02
Linha 4 - 02 02 00 03 01 02
Linha 5 - 02 05 00 06 01 03

³² OR-Library <http://people.brunel.ac.uk/~mastijb/ieb/orlib/files/jobshop1.txt>

Capítulo 6

Experimentos e Resultados

6.1 Experimentos

Este capítulo descreve os resultados do modelo computacional paralelo abordado no Simulador, onde testes e validações foram experimentados no algoritmo *busca tabu* (*kernel*) paralelo, como estudo de caso descrito no Capítulo 3.

Para testar o modelo, oito instâncias clássicas, ou *bench-mark*, disponíveis na *OR-Library*, descrito na Seção 5.8, foram utilizadas. Estas foram escolhidas pela característica de sua diversidade no tamanho de entrada para cada problema de modo a avaliar o desempenho do *kernel* no Simulador quanto à qualidade de solução, assim como, o tempo de processamento para encontrar o melhor *makespan* em um ambiente paralelo. Os *bench-marks* MT 06, MT 10 e MT 20 são citados pelos autores Muth e Thompson (1963). Os *bench-marks* La 01, LA 21, LA 26 e LA 31 são citados por Lawrence (1984) e o *bench-mark* ABZ 6 por Balas *et al* (1988).

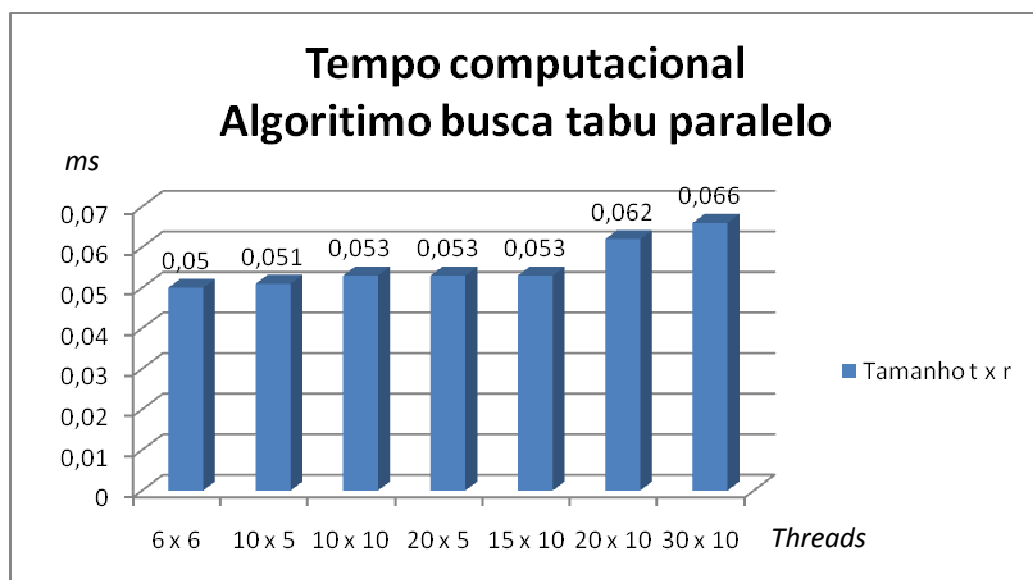


Figura 45: Tempo computacional por *threads*.

- Problema - 6 tarefas x 06 robôs- MT 06 de Muth and Thompson;
- Problema - 10 tarefas x 5 robôs - LA 01 de S. Lawrence ("Table 3, problem 1");
- Problema - 10 tarefas x 10 robôs - MT 10 de Muth & Thompson via S. Lawrence;

- Problema – 10 tarefas x 10 robôs – ABZ 6 de Balas via S. Lawrence;
- Problema - 20 tarefas x 5 robôs - MT20 de Muth & Thompson via S. Lawrence;
- Problema - 15 tarefas x 10 robôs - LA 21 de S. Lawrence (“Table 7, problem 1”);
- Problema - 20 tarefas x 10 robôs - LA 26 de S. Lawrence (“Table 8, problem 1”);
- Problema - 30 tarefas x 10 robôs - LA 31 de S. Lawrence (“Table 9, problem 1”).

As cinco primeiras instâncias são muito referenciadas na literatura (MT 06, LA 01, MT 10, ABZ 6 e MT 20) e são consideradas, relativamente, como pequenas instâncias. Entretanto, para instâncias maiores, como do tipo 20 tarefas por 20 robôs, até recentemente, não foram resolvidas otimamente (CHAO, 2008).

Para permitir a comparação entre os resultados encontrados nos experimentos realizados no Simulador, descrito na Seção 5.6, com outros métodos, utilizou-se a média de erro relativo ou “*Mean Relative Error*” (MRE). Esta média é calculada ao encontrar o melhor *makespan* de uma das soluções da instância que está sendo analisada ou “*Upper Bound*” (**UB**) desta e a solução ótima conhecida para esta instância ou “*Lower Bound*” (**LB**). Utilizou-se a seguinte fórmula para o desvio relativo:

$$MRE = \frac{(UB - LB)}{LB}$$

Tabela 8: Resultados para 08 instâncias com diferentes tamanhos de entrada.

Instância	t x r	Solução Inicial	Solução Final	Valor Ótimo (LB)	MRE	Tempo (ms)
MT06	6 x 6	60	55	55	0,00%	0,05
LA01	10 x 5	772	666	666	0,00%	0,051
MT10	10 x 10	1173	958	930	3,01%	0,052
ABZ6	10 x 10	1110	946	943	0,32%	0,041
MT20	20 x 5	1645	1380	1165	18,45%	0,05
LA21	15 x 10	1265	1192	1046	13,96%	0,053
LA26	20 x 10	1372	1355	1218	11,25%	0,062
LA31	30 x 10	1918	1845	1784	3,42%	0,066

Capítulo 7

Conclusão

7.1 Conclusão

O uso de unidades de processamento gráfico (GPU) para acelerar a computação de propósitos gerais tem despertado grande interesse pelos pesquisadores científicos nos últimos anos. No entanto, o uso desta técnica de desenvolvimento apresenta um aumento significativo na complexidade de programação em relação às soluções baseadas em uma única CPU.

Esta dissertação apresenta como contribuição a disponibilização de uma ferramenta integrada a um ambiente *web* para a elaboração de novas soluções híbridas, paralelas e seriais, na resolução do difícil e importante problema JSP, combinando a eficiência da linguagem C para um dispositivo GPU com a versatilidade da tecnologia Java.

O Simulador permite ao pesquisador desenvolver novas técnicas utilizando uma GPU fazendo uso dos recursos de *template* existentes no *kernel* deste Simulador, como por exemplo, o uso dos métodos de identificação do *caminho crítico*, a movimentação entre operações no mesmo robô e o escalonamento das operações, como apresentado no Capítulo 3. Outras facilidades importantes são encontradas neste ambiente como a infra-estrutura de relatório textual e gráfico como a documentação PDF ali existente.

No estudo de caso apresentado no Capítulo 3 foi desenvolvido um algoritmo *Busca Tabu* paralelo em pouco tempo, pela facilidade encontrada no uso destes recursos de desenvolvimento, como compilação, *debug* e relatórios de evolução do algoritmo, apresentando relativa eficiência na qualidade das soluções para pequenos e médios problemas da *Or-Library*. Alguns deles, encontrando o ótimo global. Em todas as instâncias o tempo de processamento é altamente eficiente, para um ambiente paralelo de baixo custo, ficando este por volta de algumas dezenas de micros segundos.

Nos resultados da Tabela 8, as pequenas e médias instâncias obtiveram êxito, considerando o limite de 10.000 iterações para a evolução da pesquisa de

vizinhança. Este foi um limite prático para possibilitar o compartilhamento dos recursos da placa gráfica, que atende ao servidor (*Host*), com o processamento feito pelo algoritmo da *busca tabu*.

As pequenas e médias instâncias requerem uma configuração de ambiente da GPU de apenas um bloco por *grid* e $t \times r$ *threads* por bloco. Esta configuração necessita apenas da disponibilização dos oito processadores do *Streaming Multiprocessor* (SM), como na Seção 4.7. No entanto, para instâncias maiores é exigido o uso de mais blocos (mais processadores) e mais memória *cache* (Seção 4.8.2) para realizar o processamento destas instâncias.

Com o aumento do número de blocos para as grandes instâncias foram identificados problemas de sincronização entre estes blocos do dispositivo (Seção 4.9): as *threads* possuem alta interação de dados entre elas e o sincronismo nativo (“**__syncthreads**”) existe apenas no mesmo bloco, prejudicando assim, a evolução na busca pela vizinhança para grandes problemas.

Com referência ao desempenho, a transferência de dados entre o *Host* (CPU) e o *dispositivo* (GPU) apontou ser um dos “gargalos” do processamento, pois utiliza um barramento de alta latência.

A forma de escrever o algoritmo para arquitetura paralela é fundamental para o ganho de desempenho, e é diferente de escrever o mesmo algoritmo para um processamento serial. Um algoritmo apropriado pode obter ganhos de desempenho que podem chegar a algumas ordens de grandeza.

O desempenho do Simulador depende do conhecimento da arquitetura da GPU: memória *cache*, sincronização, número de *threads* por bloco e etc. Utilizar o máximo as *caches* existentes para cada bloco e evitar utilizar a memória global do dispositivo de modo a alcançar alto desempenho, apresentou bons resultados: nas primeiras versões, utilizou-se apenas a memória Global do dispositivo e o melhor tempo de resposta do algoritmo foi de apenas alguns milissegundos (~ 6 ms) para gerar uma solução viável inicial para pequena instância.

7.2 Limitações e Trabalhos Futuros

Procurar utilizar o paralelismo entre o processamento do *Host* (CPU) e do dispositivo (GPU) é outro método interessante para grandes instâncias, pois ambos podem realizar processamentos independentes.

Outras restrições da tecnologia CUDA como funções recursivas (não permitidas), consistem em utilizar funções de geração de número aleatório no *kernel* e o *watchdog* existente no dispositivo (depois de certo tempo a GPU aborta o processamento no *kernel* e lança um erro “*launch time error*”), uso concomitantemente entre os processamentos do *kernel* e *gráfico* da placa gráfica podem ser contornados em trabalhos futuros:

- Desabilitar no sistema operacional o uso da placa gráfica;
- Utilizar outras barreiras de sincronização, através de algoritmos;
- Utilizar a lista de números aleatórios internamente no *kernel* para heurísticas estocásticas;
- Utilizar a família Geforce 400 *séries* da NVIDIA que possui mais memória cache interna (de 16 KB passou para 48KB) e mais núcleos (480 núcleos) para aumentar o desempenho do Simulador.

REFERÊNCIAS BIBLIOGRÁFICAS

- AUDE, J. S. et al. The Multiplus/Multiplex parallel processing environment. In: INTERNATIONAL SYMPOSIUM ON PARALLEL ARCHITECTURES ALGORITHMS, AND NETWORKS, 2., 1996. Beijing. **Proceedings ...** Los Alamitos: IEEE, 1996.
- APPLEGATE D. ; COOK, W. A computational study of the job shop scheduling problem. **ORSA Journal of Computing**, Baltimore, v. 3, n. 2, p. 149-156, Springer, 1991.
- BALAS, E. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. **Operations Research**, Baltimore, v. 17, n. 6, p. 941-957, Nov./Dec. 1969.
- BALAS E. ; ADAMS J. ; ZAWACK D. The shifting bottleneck procedure for job shop scheduling. **Management Science**, Linchicum, v.34, n.3, p. 391-401, Mar. 1988.
- BEASLEY, J. E. OR-Library: distributing test problems by electronic mail. **Journal of the Operational Research Society** Oxford, v. 41, n. 11, p. 1069-1072, Nov. 1990.
- BENI, G. ; WANG J. Self organizing sensory systems. In: TOU, J. T. ; BALCHEN, J. G. (Ed). **Highly redundant sensing in robotic systems**. Berlin: Springer, 1990. p. 251-262. Proceedings of NATO Advanced Workshop on Highly Redundant Sensing in Robotics Systems.
- BLAZEWICZ, J. ; ERWIN, P. ; MALGORZATA, S. The disjunctive graph machine representation of the job shop scheduling problem. **European Journal of Operational Research**, Amsterdam, v. 127, n. 2, p 317-331, Dec. 2000.
- BRUCKER, P. ; JURISCH, B. ; SIEVERS, B. A branch and bound algorithm for job-shop scheduling problem. **Discrete Applied Mathematics**, Amsterdam, v. 49, n. 1-3, p. 105-127, Mar. 1994.
- BUCK, I. ; BASTIAAN, A. – **Inventors - C/C++ Language Extensions for General-Purpose Graphics Processing Unit NVIDIA Corporation**, - Assignee, Patent Application Publication no. US 2008/0109795A1, Pub. Date May 8, 2008.
- CARLIER, J. ; PINSON, E. An algorithm for solving the job shop problem. **Management Science**, Providence, v. 35, n. 2, p.164-176, Feb. 1989.
- CHAYONG, Z .et al. Some new results on tabu search algorithm applied to the job-shop scheduling problem. In: JAZIRI, W. (Ed). **Tabu Search**. Vienna: INTEH, 2008. 278 p. Local Search Techniquesziri. ISBN 978-3-902613-34-9,
- CHONG, C. S. et al. A bee colony optimization algorithm to job shop scheduling. In: WINTER SIMULATION CONFERENCE, 38., 2006, Monterey. **Proceedings ...** Monterey: SCS, 2006.
- DORIGO, M. ; STÜTZLE, T. **The ant colony optimization metaheuristic: algorithms, applications, and advances**. Bruxelles: Universit Libre de Bruxelles, 2000. (Technical Rreport).
- FISHER, H. ; THOMPSON, G. L. Probabilistic learning combinations of local job-shop scheduling rules. In: MUTH, J. F. ; THOMPSON, G. L. (Ed). **Industrial scheduling**. Englewood Cliffs: Prentice-Hall, 1963. p. 225-251.
- GAREY, M. R. ; JOHNSON, D. S. ; SETHI, R. The complexity of flowshop and jobshop scheduling. **Mathematics of Operations Research**, Providence, v. 1, n. 2, p. 117-129, May 1976.

- GIFFLER, B. ; THOMPSON, G. L., Algorithms for solving production scheduling problems. **Operations Research**, Baltimore, v. 8, n. 4, p. 487-503, Jul./Aug. 1960.
- GLOVER, F. Future paths for Integer programming and links to artificial intelligence, **Computers and Operations Research**, New York, v. 13, n. 5, p. 533-549, May 1986.
- GLOVER, F. ; LAGUNA, M. **Tabu search**. Boston: Kluwer Academic Publishers, 1997. No 0-7923-9965-X.
- GRAY, C. ; HOESADA, M. Matching heuristic scheduling rules for job shops to the business sales level, **Production and Inventory Management Journal**, Fall Church, v. 4, p. 12-17, 1991.
- HALFHILL, T. R., **Parallel processing with CUDA, Microprocessor Report**, 2008 Disponível em: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf. Acesso em: abril 2010.
- HARUN, P. ; BAYRAKTAR, E ; EKSIOGLU, B. Tabu search: a comparative study local search techniques: In: JAZIRI, W. (Ed.). **Focus on tabu search**. Viena: INTEH, 2008. 278 p. ISBN 978-3-902613-34-9.
- JOHNSON, S. M. Optimal two- and three-stage production schedules with setup times included. **Naval Research. Logistic. Quarterly**, New York, v. 1, n. 1, p. 61-68, 1954.
- KANTER, D. **NVIDIA's GT200: inside a parallel processor**, 09-08-2008. Disponível em: <http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1>, Acesso em: out. 2008.
- LAWRENCE S. **Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques**. Graduate school of industrial administration, Carnegie Mellon University Pittsburgh, 1984.
- LOURENÇO, H. R. Job shop scheduling: computational study of local search and large-step optimization methods. **European Journal of Operational Research**, Amsterdam, v., 83, n. 2, p. 347-364, June 1995.
- MURPHY, M. **NVIDIA's Experience with Open64**. Jan. 2008. Disponível em: <http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc> Acesso em: abril 2010.
- MUTH, J. F. ; THOMPSON, G. L. **Industrial scheduling**. Englewood Cliffs: Prentice Hall, 1963.
- MARTIJN, C. **Schut Scientific Handobook for Simulation of Collective Intelligence**. Version: 2, February 2007. Disponível em: <http://www.scribd.com/doc/244626/Scientific-Handbook-for-the-Simulation-of-Collective-Intelligence>. Acesso em: abril 2010.
- NVIDIA Corporation. **CUDA Programming Guide**. Version 2.0, 2008.
- NICKOLLS, J. R. et al. **Inventors - Virtual architecture and instruction set for parallel thread computer**. NVIDIA Corporation, - Assignee, Patent Application Publication no. JP2008276740 (A), Pub. Date Nov 13, 2008.
- NICHOLSON, T. A. J. **Optimization in industry**. London: Longman Press, 1971. (Optimization techniques, v. 1).
- PINEDO, M. **Scheduling theory, algorithms, and systems**. Englewood Cliffs: Prentice-Hall, 1995.

- PINSON, E. The job shop scheduling problem: A concise survey and some recent developments. In: CHRÉTIENNE, P. et al (Ed.). **Scheduling theory and its application**, West Sussex: John Wiley and Sons, 1995. chap. 13, p. 277-293.
- HUANG, Q. et al. GPU as a General Purpose Computing Resource. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING, APPLICATIONS AND TECHNOLOGIES, 9., 2008, Dunedin. **Proceedings ...** Los Alamitos: IEEE, 2008. p. 151-158.
- RIPON, K. S. N. Hybrid evolutionary approach for multi-objective job shop scheduling problem. **Malaysian Journal of Computer Science**, Kuala Lumpur, v. 20, n. 2, p. 183-198, 2007.
- ROY B. ; SUSSMANN B. **Les problemes d'ordonnancement avec contraintes disjonctives**,. Paris: SEMA, 1964. (Note D.S. n. 9)
- ŠEDA, M. **Mathematical models of flow shop and job shop scheduling problems**. [S.l.]. World Academy of Science, Engineering and Technology, 31, 2007.
- RYOO, S. et al. Program optimization space pruning for a multithreaded GPU In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 6., 2008. BOSTON. **Proceedings ...** New York, 2008.
- SHOUICHI, M. ; YAMADA, S. An empirical performance evaluation of a parameter-free genetic algorithm for job-shop scheduling problem. In IEEE CONGRESS ON EVOLUTIONARY COMPUTATION, 2007, Singapore. **Proceedings ...** Los Alamitos, 2007. p. 3796-3803.
- TAILLARD, É. D. Parallel tabu search techniques for the job shop scheduling problem. **ORSA Journal on Computing**, Baltimore, v. 6, n. 2, p. 108–117, Spring 1994.
- VAESSENS, R. J. M. ; AARTS, E. H .L.; LENSTRA, J. K. Job shop scheduling by local search, **INFORMS Journal on Computing**, Hanover, v. 8, n. 3, p. 302-317, Summer. 1996.
- YAMADA, T .; NAKANO R. Genetic algorithms for job-shop scheduling problems. IN: UNICOM SEMINAR, 1997, London. **Proceedings of Modern Heuristic for Decision Support**, London: 1997. p. 67-81.
- YAMADA, T. ; NAKANO R., Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search. In: OSMAN, I. H. ; KELLY, J. P. **Meta-heuristics: theory & applications**. Norwell: Kluwer Academic Publishers, 1996. p. 237–248.
- ZHIRNOV, V. V. et al. Limits to binary logic switch scaling – a gedanken model. **Proceedings of the IEEE**, New York, v 91, n. 11, p. 1934-1939, Nov. 2003.