2.
Fundamentação Teórica

2.1.
Introdução

Neste capítulo são apresentados os conceitos fundamentais sobre Computação Evolutiva, Programação Genética Linear, Programação Genética Linear com Inspiração Quântica (PGLIQ) — que é a técnica de Inteligência Computacional empregada como base nesta dissertação —, Computação de Alto Desempenho, Visualização Gráfica e Robótica Evolutiva.

2.2.
Computação Evolutiva

Computação Evolutiva é um termo aplicado a uma grande gama de metodologias relacionadas à resolução de problemas computacionais que são baseados ou inspirados em processos biológicos. As principais ideias envolvidas têm uma história surpreendentemente longa, com Alan Turing propondo um método de busca genética evolutiva, já em 1948, e os experimentos de computadores ativos aparecendo na década de 1960 [8].

Os métodos de computação evolutiva tais como buscas aleatórias, algoritmos genéticos e métodos de enxame de partículas, podem ser considerados um subconjunto de um conjunto maior de algoritmos de otimização estocástica. Esta família de técnicas compartilha a característica comum de utilizar aleatoriedade em algum aspecto de sua progressão através do espaço de soluções. Este recurso busca melhorar a eficácia global do algoritmo em comparação com técnicas não estocásticas, como evitar ficar preso em ótimos locais ou superar os efeitos do ruído no processo da otimização [9].

No entanto, é difícil generalizar entre estes métodos dado que seu desempenho é muitas vezes específico do problema e altamente sensível aos parâmetros de ajuste do algoritmo [10] [11]. Uma representação gráfica que mostra vários métodos algorítmicos e as relações entre eles é apresentada na Figura 1.

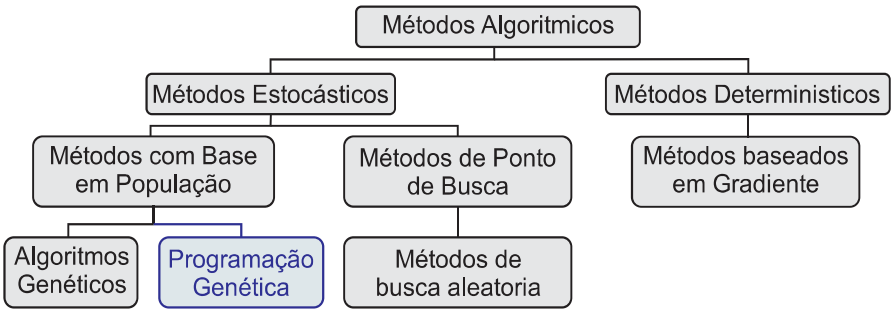


Figura 1. Representação gráfica de métodos algorítmicos e relacionamentos

Alguns dos elementos comumente usados em computação evolutiva são:

Individuo: Um indivíduo é um ser que caracteriza a sua própria espécie. O indivíduo é um cromossomo e é o código de informação em que o algoritmo funciona.

População: Um conjunto de indivíduos (cromossomos) é chamado de população.

Função de aptidão: Função que informa o quão bom é um indivíduo para resolver um problema.

2.2.1. Programação Genética Linear

Nos últimos anos, diferentes variantes de programação genética têm surgido. Todas seguem a ideia básica da programação genética de evoluir automaticamente programas de computador. Três formas básicas de representação podem ser distinguidas por programas genéticos. Além das representações de árvores tradicionais, há as representações lineares [12] e em forma de grafo [13] [14]. Os programas de árvores usados na programação genética [15] correspondem a expressões (sintaxe de árvores) de uma linguagem de programação funcional. Esta abordagem clássica também é conhecida como programação genética baseada em árvore, na qual as funções estão localizadas nos nós internos, enquanto as folhas da árvore mantêm os valores de entrada ou constantes.

Instrução 1	Instrução 2	...	Instrução N

Figura 2. Representação típica de PG linear para programas.

Em contrapartida, a PGL é uma variante da programação genética que evolui sequências de instruções de uma linguagem de programação imperativa ou de uma linguagem de máquina. Cada programa é uma sequência de instruções, tal como mostrado na Figura 2. O número de instruções pode ser fixo, significando que todos os programas da população têm o mesmo comprimento, ou variável, o que significa que diferentes indivíduos podem ter diferentes tamanhos.

2.2.2. Demes

A noção de *demes* foi introduzida por Wright em 1943 como uma descrição de um fenômeno em evolução natural, onde uma população animal é dividida em subpopulações. Em algoritmos evolutivos o termo é utilizado como a unidade de um conjunto de populações. Estas populações evoluem de forma independente tentando otimizar o mesmo objetivo [16]. Normalmente, este conjunto de *demes* está estruturado em forma de anéis ou rede de vizinhança (Figura 3). Assim, cada deme troca alguns indivíduos com seus vizinhos em cada intervalo de gerações. Esta troca de indivíduos é chamada de migração. [17]

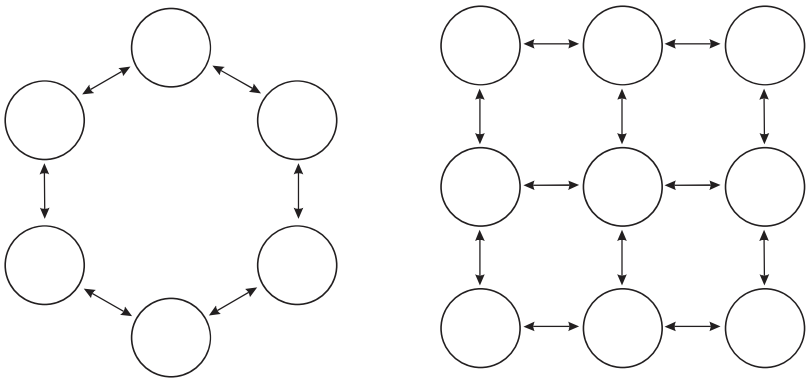


Figura 3. Modelo de *demes* como anéis (à esquerda) e rede de vizinhança (à direita).

Alguns dos parâmetros de demes que são utilizados na configuração de um algoritmo evolutivo:

- **Número de demes:** Número de populações que evoluem o mesmo objetivo.
- **Tamanho da subpopulação:** Numero de indivíduos em cada *deme*. O número total de indivíduos é:

$$\#\text{Total de Indivíduos} = \text{Numero de } demes \times \text{Tamanho da subpopulação}$$

- **Intervalo de gerações para migração:** Número de gerações em que os *demes* evoluem isoladamente antes de migrar indivíduos entre eles.
- **Taxa de migração:** porcentagem da subpopulação de indivíduos que migrarão de um *deme* para outro.

Os *demes* são utilizados para reduzir a possibilidade de a evolução acabar em um mínimo local, além de serem altamente paralelizáveis, aumentando assim a eficiência do algoritmo. Este "*speed-up* super-linear" tem sido observado em simulações de algoritmos evolucionários.

2.3. Programação Genética Linear com Inspiração Quântica (PGLIQ)

Esta dissertação se concentra em desenvolver uma extensão da versão de PGLIQ proposta em [2]. Portanto, nesta seção são apresentados os detalhes de sua operação. O desempenho médio da PGLIQ é superior ao dos modelos de referência de PG. Este desempenho é baseado em desenvolver uma nova abordagem denominada Computação com Inspiração Quântica, cujo objetivo é criar algoritmos clássicos (executados em computadores clássicos) que tirem proveito dos princípios da mecânica quântica para melhorar seu desempenho.

2.3.1. Plataforma

A PGLIQ utiliza algumas instruções relativas à Unidade de Ponto Flutuante (*Floating Point Unit* – FPU) da CPU, as quais, por sua vez, podem trabalhar com dados da memória principal e/ou dos oito registradores da FPU (*ST(i)*, onde i = 0, 1, ... , 7). Um programa em código de máquina evoluído pela PGLIQ representa

uma solução. Este programa lê os dados de entrada a partir da memória principal, os quais são compostos pelas variáveis de entrada do problema e, opcionalmente, por algumas constantes fornecidas pelo usuário. Estas entradas podem ser representadas por um vetor, conforme exemplificado a seguir:

$$\mathrm{I} = (\mathrm{V}[0], \mathrm{V}[1], 1, 2, 3) \tag{2.1}$$

onde *V [0]* e *V [1]* contêm os valores das duas entradas do problema, e onde 1, 2 e 3 são os valores das três constantes.

2.3.2. Representação

A PGLIQ é baseada nas entidades da seguinte forma: o cromossomo do "indivíduo quântico", que representa a superposição de todos os programas possíveis para o espaço de busca predefinido, é observado para se gerar o cromossomo do "indivíduo clássico", a partir do qual o programa em código de máquina é finalmente gerado. Ou seja, o cromossomo de um indivíduo clássico é a representação interna de um programa em código de máquina.

2.3.3. Indivíduo Clássico

O modelo representa internamente as funções por um "*token* de função" (TF), que pode assumir valores inteiros de 0 a (f - 1), de forma a representar de maneira única cada uma das f funções da PGLIQ. A Tabela 1 mostra um exemplo de conjunto de funções definido para um dado problema, onde os valores dos *tokens* são representados na base hexadecimal. Conforme mencionado, um programa é internamente representado pelo cromossomo do indivíduo clássico, que pode ser representado por uma estrutura com (L x 2) *tokens*, conforme mostrado na Figura 4, onde: L é o comprimento máximo do programa (em número de instruções); cada linha representa uma instrução i ($1 \leq i \leq \mathrm{L}$); a coluna da esquerda contém os valores dos *tokens* de função (TF_i) e a da direita, os valores dos seus respectivos *tokens* de terminal (TT_i).

Instrução	Descrição	Valor do token
NOP	Nenhuma operação	0
FADD *m*	$ST(0) \leftarrow ST(0) + m$	1
FADD ST(0), ST(***i***)	$ST(0) \leftarrow ST(0) + ST(i)$	2
FADD ST(***i***), ST(0)	$ST(i) \leftarrow ST(i) + ST(0)$	3
FSUB *m*	$ST(0) \leftarrow ST(0) - m$	4
FSUB ST(0), ST(***i***)	$ST(0) \leftarrow ST(0) - ST(i)$	5
FSUB ST(i), ST(0)	$ST(i) \leftarrow ST(i) - ST(0)$	6
FMUL *m*	$ST(0) \leftarrow ST(0) \times m$	7
FMUL ST(0), ST(***i***)	$ST(0) \leftarrow ST(0) \times ST(i)$	8
FMUL ST(***i***), ST(0)	$ST(i) \leftarrow ST(i) \times ST(0)$	9
FXCH ST(***i***)	$ST(0) \leftrightarrow ST(0)$	A

Tabela 1. Exemplo de um conjunto de funções e seus *tokens*

A ordem de execução do programa é da primeira até a última linha. Cada linha é definida como sendo um "gene".

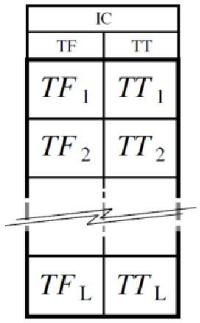


Figura 4. Cromossomo de um indivíduo clássico.

2.3.4. Indivíduo Quântico

O Indivíduo Quântico é inspirado em sistemas quânticos multiníveis. A unidade básica de informação adotada pela PGLIQ é o *qudit*. Esta informação pode ser descrita por um vetor de estado em um sistema mecânico quântico de *d* níveis, o qual equivale a um espaço vetorial *d*-dimensional, onde *d* é o número de estados

nos quais o *qudit* pode ser medido. Isto é, *d* representa a cardinalidade do *token* que terá seu valor determinado pela observação do seu respectivo *qudit*. O estado de um *qudit* é uma superposição linear dos *d* estados e pode ser representado pela equação 2.1.

$$|\psi\rangle = \sum_{i}^{d-1} \alpha_i |i\rangle \qquad ((2.2)$$

Na qual o valor de $|\alpha|^2$ representa a probabilidade *p* de que o *qudit* seja encontrado no estado *i* quando é observado. A normalização unitária do estado garante: $\sum_{i=0}^{d-1} |\alpha_i|^2 = 1$.

2.3.5. Programas Evoluídos

Cada programa evoluído pela PGLIQ é um programa composto pelos três seguintes segmentos: Cabeçalho (*header*), Corpo e Rodapé (*footer*). Nem o cabeçalho, nem o rodapé são afetados pelo processo evolutivo. O rodapé transfere o conteúdo de ST(0) para V [0], uma vez que esta é a saída predefinida do programa, e reinicia a FPU.

2.3.6. Avaliação de um Indivíduo Clássico

Este processo se inicia com a geração de um programa em código de máquina a partir do indivíduo clássico a ser avaliado. Conforme mostrado na seção 2.3.4, neste processo, o cromossomo do indivíduo clássico é percorrido sequencialmente, gene por gene, *token* por *token* (de função e de terminal), gerando serialmente o código de máquina do corpo do programa relativo àquele indivíduo clássico.

2.3.7. Operador Quântico

O operador quântico aqui proposto atua diretamente nas probabilidades p_i de um *qudit*, satisfazendo a condição de normalização: $\sum_{i=0}^{d-1} |\alpha_i|^2 = 1$, onde d é a

cardinalidade do *qudit* e $|\alpha_i|^2 = p_i$. Este operador aqui é denominado "operador P", e representa a funcionalidade de uma porta quântica, efetuando rotações no vetor que representa o estado $|\psi\rangle$ de um *qudit* em um espaço vetorial *d*-dimensional.

O operador P funciona em dois passos básicos. Primeiramente, incrementa uma dada probabilidade do *qudit*, da seguinte maneira:

$$p_i \leftarrow p_i + s(1 - p_i) \tag{2.3}$$

onde *s* é um parâmetro denominado "tamanho de passo", que pode assumir qualquer valor real entre 0 e 1. O segundo passo é o ajuste dos valores de todas as probabilidades do *qudit* de forma a satisfazer a condição de normalização. Portanto, este operador pode ser representado pela função:

$$q' = P(q, i, s) \tag{2.4}$$

que modifica o estado do *qudit* q, resultando no *qudit* q'. Esta função incrementa o valor da probabilidade p_i do qudit q de uma quantidade que, por sua vez, é diretamente proporcional ao valor do tamanho de passo *s*.

2.3.8. Estrutura e Funcionamento do Modelo

Com relação à estrutura, a PGLIQ possui uma população híbrida, composta por duas populações, uma quântica e outra clássica, ambas possuindo o mesmo número *M* de indivíduos. Também possui *M* indivíduos clássicos auxiliares C_{obs}, que resultam das observações dos indivíduos quânticos Q*i*, onde $1 \leq i \leq M$. No caso exemplificado pelo diagrama da Figura 5, $M = 4$. Além da estrutura, o mesmo diagrama apresenta, enumeradas, as quatro etapas básicas que caracterizam uma "geração" do modelo, as quais são descritas a seguir.

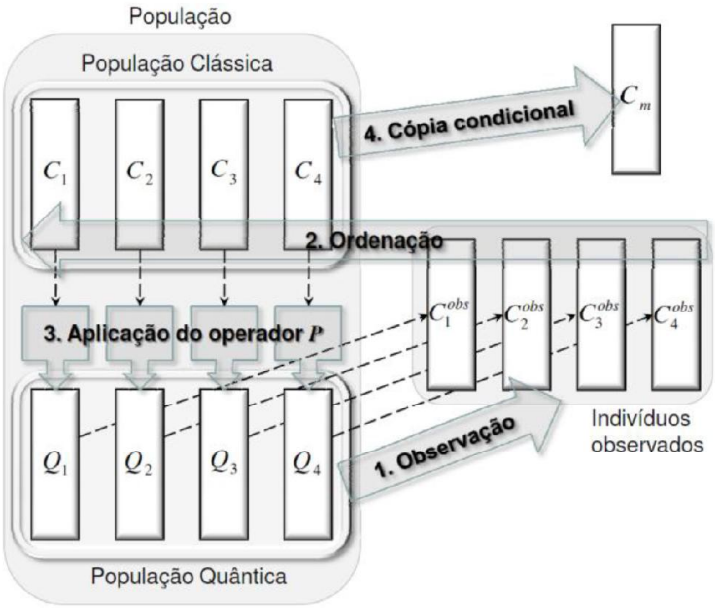


Figura 5. Diagrama descritivo básico do modelo PGLIQ.

1. Cada um dos *M* indivíduos quânticos é observado uma vez, resultando nos *M* indivíduos clássicos C_{obs}.
2. É efetuada a ordenação conjunta dos indivíduos da população clássica e dos indivíduos observados quanto às suas avaliações. Como resultado, os *M* melhores indivíduos clássicos dentre os 2*M* avaliados são mantidos na população clássica, ordenados do melhor para o pior, de C_1 paraC_M. Os demais *M* indivíduos clássicos permanecem armazenados e ordenados, do melhor para o pior, de C_1^{obs} para C_M^{obs}.
3. O operador *P* é aplicado a cada indivíduo da população quântica, tomando como referência seus indivíduos correspondentes na população clássica. É nesta etapa que ocorre efetivamente a evolução, uma vez que, a cada nova geração, a aplicação do operador aumenta a probabilidade de que a observação dos indivíduos quânticos gere indivíduos clássicos mais parecidos com os melhores encontrados até então.
4. Caso algum dos indivíduos da população clássica, avaliados na geração atual, seja melhor que o melhor indivíduo clássico avaliado até então (em relação às gerações anteriores), uma cópia do mesmo é armazenada em C_M, o melhor indivíduo clássico encontrado pelo algoritmo até então.

Um diagrama de fluxo da PGLIQ é apresentado na Figura 6, o algoritmo é dividido em quatro etapas que serão utilizadas nos capítulos posteriores: Etapa de Inicialização, Etapa de Evolução, Etapa de Validação e Etapa de *Reset*.

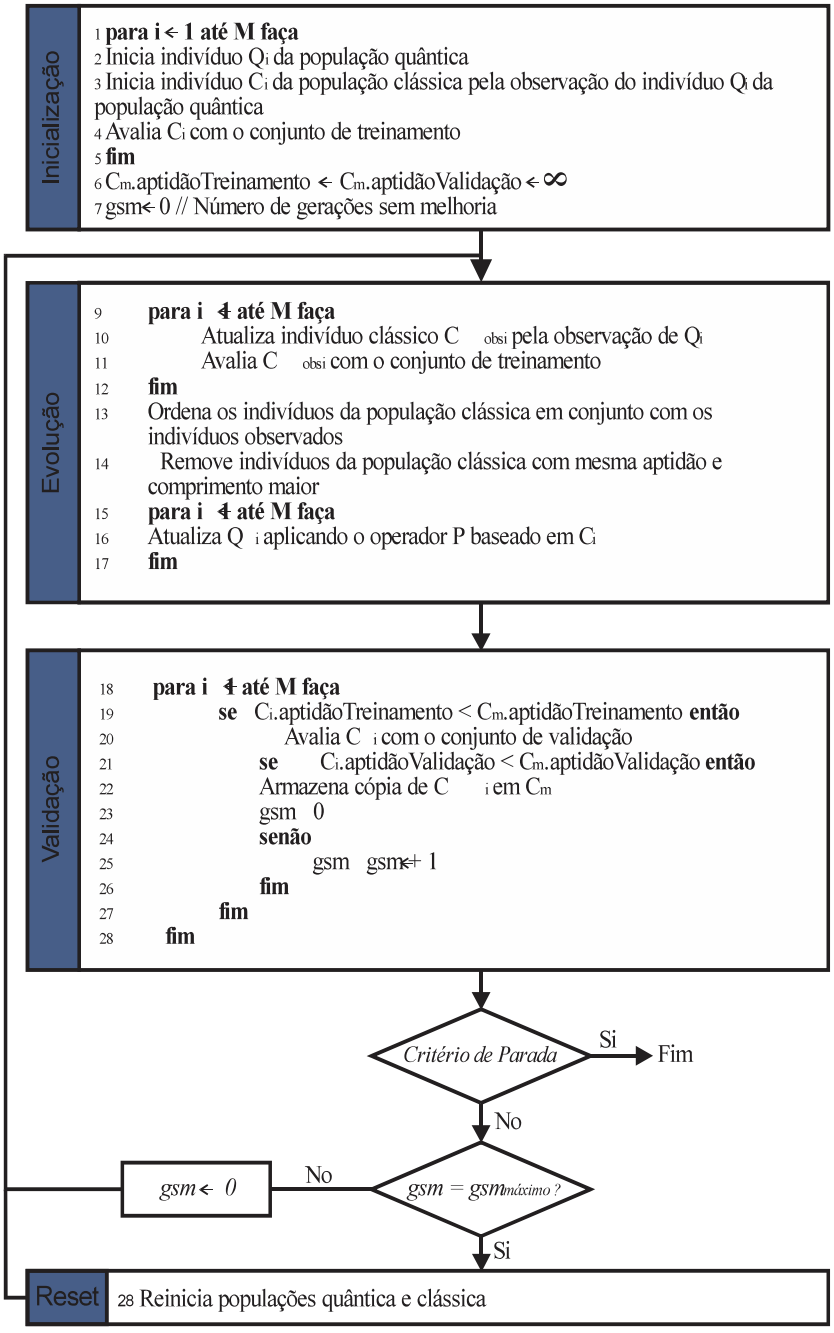


Figura 6. Diagrama de fluxo da PGLIQ

2.4. Computação de Alto Desempenho

Nas arquiteturas *multi-core* e *many-core* (como as unidades de processamento gráfico - GPUs), são utilizados conceitos importantes de computação de alto desempenho, os quais nesta seção são explicados juntamente com as técnicas e bibliotecas de alto desempenho utilizadas.

2.4.1. Introdução: GPUs vs CPUs

As CPUs tradicionais tendem a ter 4-8 processadores que são apropiados para executar grandes operações em pequenos conjuntos de dados, mas muito lentos na comunicação entre os processadores múltiplos. CPUs são projetados para serem capazes de mudar ativamente a tarefa em tempo real, o que os torna muito bons em, por exemplo, executar, simultaneamente, um *web-browser*, uma decodificação de vídeo e uma partida de jogo de vídeo.

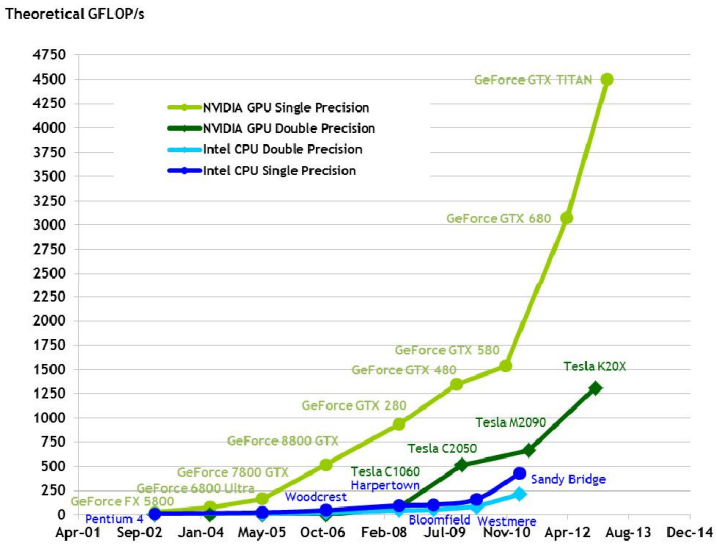


Figura 7. Comparação de desempenho de GPUs *vs* CPU. Desempenho da GPU continua a aumentar a um ritmo muito rápido.

No entanto, esta flexibilidade exige um grande número de ciclos para alternar entre tarefas, e uma grande quantidade de *cache* para armazenar tarefas

parcialmente concluídas. As Unidades de Processamento Gráfico, ou GPUs, renunciam à flexibilidade de CPUs em favor da capacidade de processamento.

Reduzindo o tamanho da *cache* e utilizando paralelismo *Single Instruction Multiple Threads* (SIMT) permite aos fabricantes de GPU combinar centenas de processadores em um único *chip*. A fim de fornecer dados suficientes para manter centenas de processadores ocupados, GPUs também têm um grande canal de dados entre os processadores e DRAM. Todos esses recursos são escolhidos para criar um processador matemático que se destaca em operações, onde cada processador opera em dados invisíveis para os outros processadores. Estas características dão às GPUs uma significativa vantagem de desempenho em ponto flutuante sobre os CPUs, como mostrado na Figura 7 [18]. As vantagens que os GPUs têm sobre os CPUs para computação científica incluem:

- Maior desempenho por custo unitário.
- Maior desempenho por *Watt*.
- Mais fácil de atualizar (tecnologia que melhora constantemente).
- GPUs ainda estão melhorando com a Lei de Moore.

Embora essas vantagens sejam muito promissoras, existem também várias desvantagens para a computação em GPU:

- Aumento da complexidade do código
- Espaço de memória menor
- Menor *cache*
- Comunicação lenta entre CPU e GPU.
- Algoritmos mais dependentes da configuração de *hardware*.

2.4.2. Conceitos gerais de computação paralela

a) Terminologia

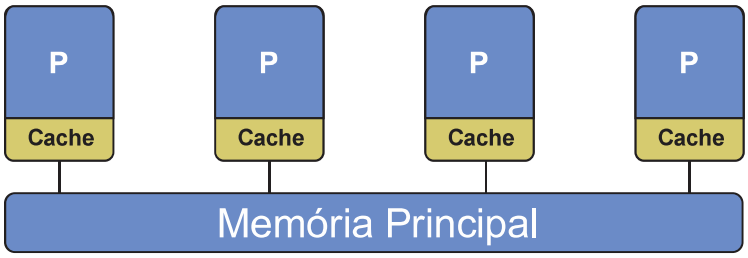
Os termos de alto desempenho mudam conforme evolui a tecnologia. Para evitar qualquer ambiguidade no uso de termos nesta dissertação, estes são relacionados didaticamente no parágrafo seguinte:

"Um *cluster* é constituído por um certo número de **nós**, cada um dos quais contém um certo número de **núcleos** (*cores*). Um programa que utliza um *cluster*, escolhe quantos **processos** devem ser colocado em cada **nó**. Cada **processo** inicializa um número de ***threads***".

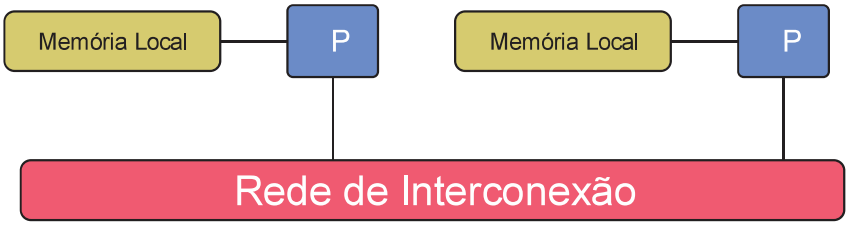
Estes **processos** também chamados **processos MPI** são explicados na seção 2.4.4. letra c.

b) Arquiteturas

Existem duas classes distintas de sistemas com múltiplos processadores: os multiprocessadores com memória compartilhada, que possuem uma unidade de memória acessível por todos os processadores, e os multiprocessadores com memória distribuída, onde os processadores têm apenas acesso às suas respectivas memórias locais (Figura 8). As técnicas de paralelização empregadas nestas duas classes são diferentes. Nesta dissertação, abordaremos os aspectos de programação para multiprocessadores com ambas as classes de memória.



(a) Multiprocessadores com memória compartilhada.



(b) Multiprocessadores com memória distribuída.

Figura 8. Arquiteturas de sistema de multiprocessadores.

c) Paradigmas de Programação Paralela

A paralelização do algoritmo, em termos de blocos de instruções executadas paralelamente, pode ser classificada em: homogênea e heterogênea.

b.1) Paralelização homogênea: A paralelização homogênea (Figura 9) envolve a execução paralela do mesmo código sobre elementos múltiplos de dados. Esta paralelização é efetuada em "*loops*", devendo para tanto ser realizada uma análise de dependência de dados no seu código. A execução de cada iteração deve ser completamente independente de qualquer outra.

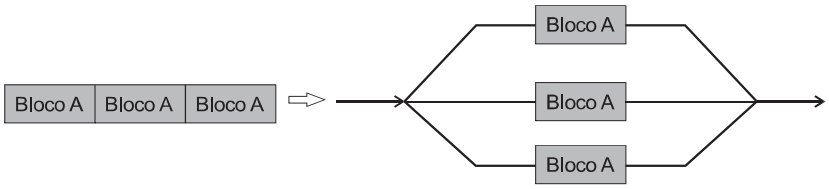


Figura 9. Paralelização homogênea.

b.2) Paralelização heterogênea: A paralelização heterogênea (Figura 10) envolve a execução paralela de códigos múltiplos sobre dados múltiplos. Esta paralelização é possível quando a tarefa a ser realizada pelo algoritmo pode ser particionada em múltiplas subtarefas, sendo que cada uma, em um ambiente paralelo, poderia ser executada por um processador.

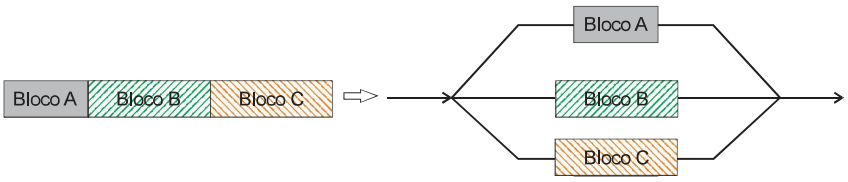


Figura 10. Paralelização heterogênea.

d) Granularidade

A granularidade reflete a distribuição do trabalho do algoritmo em subtarefas independentes executadas paralelamente. A granularidade pode ser fina ("*fine*")

ou grossa ("*coarse*"). Uma medida de granularidade é a quantidade de processamento envolvido em uma subtarefa. Por exemplo, se uma subtarefa faz uma única atribuição ao dado pode ser considerada como de granularidade fina. Já se envolve a execução de múltiplas atribuições ou mesmo chamadas de subrotinas pode ser considerada de granularidade grossa. Cabe ressaltar que devido ao custo inerente da paralelização, como, por exemplo, o custo referente à sincronização, a granulação influi no desempenho do algoritmo.

2.4.3. Modelos de programação paralela utilizada

Considerando os métodos de paralelização e o fator da granularidade podem-se classificar quatro modelos de programação paralela principais.

Paralelismo Homogêneo de granulação fina: Envolve a execução de um mesmo código que contém pouca manipulação do dado sobre múltiplos elementos de dados.

Paralelismo Homogêneo de granulação grossa: Envolve a execução de um mesmo código que contém, por exemplo, múltiplas atribuições ou mesmo chamadas de subrotinas sobre múltiplos elementos de dados.

Paralelismo Heterogêneo de granulação fina: Envolve a execução de códigos distintos com pouca manipulação do dado sobre múltiplos dados.

Paralelismo Heterogêneo de granulação grossa: Envolve a execução de códigos distintos com muita manipulação do dado realizando múltiplas atribuições ou até mesmo chamadas de subrotinas sobre múltiplos dados.

2.4.4. Ferramentas de programação paralela utilizadas

a) Computação com GPU:

As GPUs são dispositivos multi-processadores que foram projetados especificamente para computação gráfica e eram difíceis de programar antes do advento de CUDA e OpenCL. Ao contrário das unidades centrais de processamento (CPUs), que são projetadas para ter um bom desempenho na execução de tarefas

em série, as GPUs são concebidas como motores de computação numérica paralela e têm bom desempenho na computação e no processamento de dados.

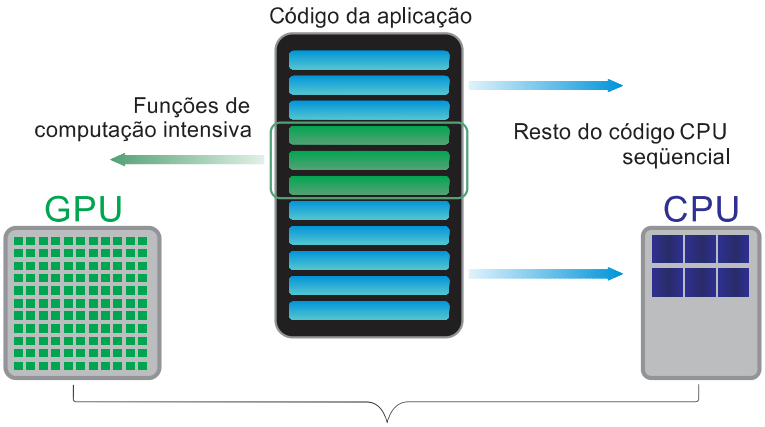


Figura 11. Aplicação conjunta CPU-GPU.

Dado que os modelos de programação em GPU (CUDA e OpenCL) são projetados para suportar e executar aplicações conjuntas CPU-GPU (Figura 11), na maioria das aplicações são utilizados ambos, executando as partes sequenciais na CPU e as partes onde a computação númerica é intensa nas GPUs.

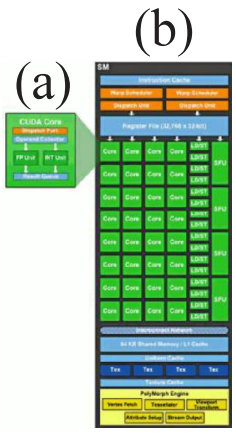


Figura 12. (a) Streaming Processors e (b) Streaming Multiprocessor

A arquitetura GPU é fortemente modular, a fim de obter a maior escalabilidade. Na nomenclatura da NVIDIA, os blocos base são os *Streaming Processors* (SP) ou núcleos CUDA, cada um equipado com uma unidade aritmética lógica (ALU) e uma unidade de ponto flutuante adicional (FPU). Uma matriz de

SPs mais duas Unidades Funcionais Especiais (SFUs) dedicadas a realizar instruções transcendentais como raiz quadrada, recíproca, seno e cosseno, formam o *Streaming Multiprocessor* (SM), que se mostra na Figura 12.

O sistema de computação está composto por um *Host*, que é uma CPU tradicional, e um ou mais dispositivos (*Device*). Um programa CUDA deve ser estruturado em fases: as fases com pouco ou nenhum paralelismo de dados são executados no *host*, enquanto as fases que apresentam enorme paralelismo de dados são executadas no dispositivo. O programa fornece um único código fonte englobando tanto *Host* e código do dispositivo, e o compilador *nvcc (**N**VIDIA **C**UDA **C**ompiler)* separa os dois.

Em CUDA, *Host* e *Device* têm espaços de memória separados. A fim de executar a função do *Device*, chamado *kernel*, o programador precisa alocar memória no dispositivo, transferir dados, e, uma vez que a execução do dispositivo seja concluída, liberar memória do dispositivo. A função do *kernel* gera um grande número de *trhreads*, organizados em uma hierarquia de dois níveis. No nível superior, os *trhreads* formam uma *grid* bidimensional que consiste em um ou mais blocos de *thread*. Cada bloco de *thread* por sua vez é organizado como uma matriz tridimensional de *threads*, mas nem todas as aplicações utilizarão todas as três dimensões em um bloco de *threads*.

Finalmente, as GPUs, ao não executar bem algoritmos seqüenciais, a criação de algoritmos paralelos e a estrutura do fluxo da arquitetura GPU, requer um grande cuidado. Além disso, as transferências *Host-Device* (e vice-versa) e acessos a muitos espaços de memória disponíveis *on-Device* devem ser uma preocupação primordial para o desenvolvedor GPU obter bom desempenho.

b) OpenMP:

Open Message Passing ou *Message Passing Interface* OpenMP é uma implementação de *multi-threading*, um esquema de execução paralela em que o *thread* mestre (uma série de instruções executadas consecutivamente) "*forks*" um determinado número de escravos *threads* e uma tarefa é dividida entre eles [19]. Ela pode ser utilizada em C/C++/Fortran. O programa é normalmente executado por vários *threads* independentes que compartilham dados, mas também pode ter algumas zonas adicionais de memória privada [20]. OpenMP está em alto nível de

abstração o que facilita o desenvolvimento de aplicações paralelas a partir da perspectiva do desenvolvedor. Em algumas aplicações, OpenMP pode esconder e implementa por si só os detalhes como particionamento de carga de trabalho, gestão de trabalho, comunicação e sincronização.

c) **OpenMPI**:

É uma biblioteca de troca de mensagens que segue o padrão *Message Passing Interface* (MPI) projetada para funcionar em uma ampla variedade de computadores paralelos. Ela pode ser especificada em C/C++/Fortran. Enquanto OpenMP funciona em sistemas de multiprocessamento simétrico (SMP), a MPI funciona tanto em SMP como de memória distribuída. É portátil, não há necessidade de modificar o código-fonte ao portar a aplicação para uma plataforma diferente que suporta (e é compatível com) o MPI padrão.

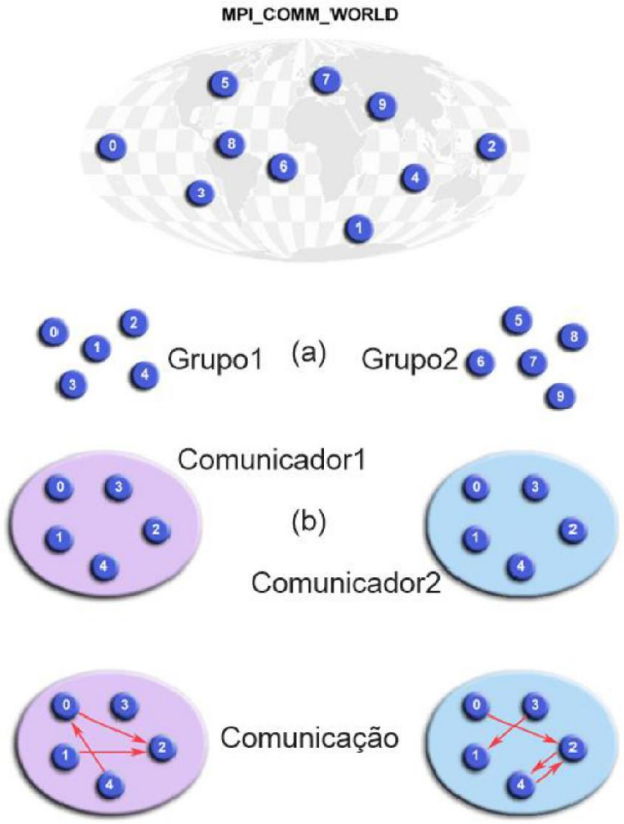


Figura 13. Modelo MPI. (a) Grupos MPI. (b) Comunicadores MPI.

MPI utiliza objetos chamados comunicadores e grupos para definir qual conjunto de processos podem se comunicar uns com os outros. A Figura 13 mostra

um exemplo em (a) onde os grupos *Grupo1* e *Grupo2* são criados desde o grupo principal *World* e em (b) os comunicadores *Comunicador1* e *Comunicador2* têm dominio dentro de um determinador grupo (neste exemplo, Grupo1 e Grupo2, respectivamente). A maioria das rotinas MPI exige a especificação de um comunicador como um argumento. Cada processo tem seu próprio identificador inteiro único atribuído pelo sistema quando o processo é inicializado, chamado *rank*. Os *ranks* são usados pelo programador para especificar a origem e o destino das mensagens. Os programas MPI sempre trabalham com processos, mas os programadores geralmente se referem aos processos como processadores.

Normalmente, para ter o máximo de desempenho, cada CPU (ou núcleo em uma máquina *multi-core*) será atribuído apenas a um único processo.

d) **Programação Híbrida MPI + OpenMP:**

MPI é voltado para a comunicação entre nós de memória distribuída e OpenMP é usado para paralelismo refinado nos processadores dentro de nós com memória compartilhada. O modelo de programação híbrida MPI + OpenMP é uma estrutura multi-nível interna de aplicações em grande escala e se encaixa com um modelo de máquina hierárquico (Figura14).

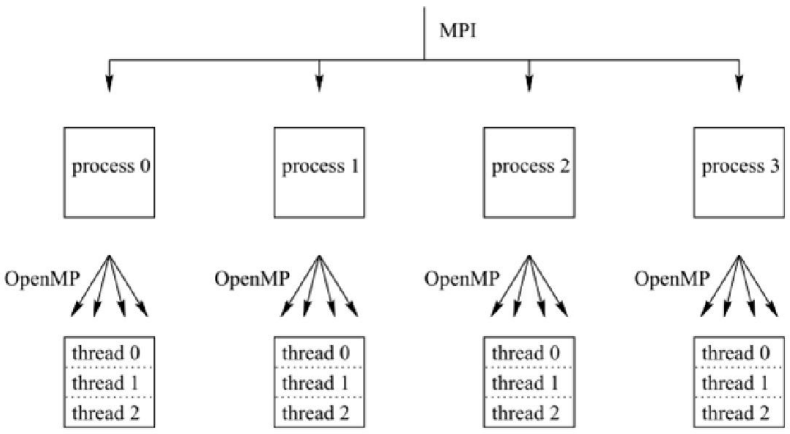


Figura 14. Modelo Hibrido MPI + OpenMP.

Este modelo de programação híbrida é para atingir o paralelismo em dois níveis, paralelismo de granulação grossa entre nós e paralelismo refinado dentro dos nós. As vantagens deste modelo são, principalmente, não utilizar troca de

mensagens dentro dos nós, não ter problemas de topológia, reduzir o tráfego de transferência de memória etc. Quando o *cluster*, que é uma estação de trabalho com modelo de programação híbrida MPI + OpenMP, tem *m* nós, cada processador possui muitas CPUs que têm *n* núcleos. Para conseguir plena utilização de recursos de computação paralela, se é usado o modelo de programação MPI, o programa deve criar os *m*n* processos quando está em execução e cada processo não é finalizado até o final do programa. Se o modelo de programação usado é híbrido MPI + OpenMP, o programa só cria *m* processos, que derivam *n threads* (cada *thread* em um núcleo de um nó). A tarefa de cada processo será dividida em *n* partes, que serão dadas a cada *thread*. O *thread* não precisa de se comunicar, o que irá reduzir os gastos de comunicação. O *thread* vai sair ou suspenderá após a execução, de forma que apenas os *m* processos estarão vivos no procedimento que está sendo executado. O algoritmo de programação híbrida é criado adicionando a declaração de compilação guia, que é #pragma omp parallel num_threads (N) (N representa o número de threads) a frente do *bucle* for dentro do algoritmo paralelo MPI. O algoritmo basico de programação híbrida é mostrado na Figura 15.

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
ierr=MPI_Init(&argc,&argv[]);
ierr=MPI_Comm_rank(...,&rank);
ierr=MPI_Comm_size(...,&size);
<configure shared mem, compute & Comm>
#pragma omp parallel for
for(i=0; i<n; i++){
<work>
}
<compute & communicate>
ierr=MPI_Finalize();
```

Figura 15. Algoritmo de Programação Híbrida

2.4.5. Níveis de Paralelismo

A paralelização em CPU é feita dividindo as tarefas de um conjunto de *threads* em diferentes processadores (as cores em computadores *multi-cores*), utilizando um API de comunicação e sincronização como *Message Passing Interface* (MPI) que permite comunicar os *threads*. Esta rede de *threads* tem um nó

principal, geralmente nó 0, que gerencia a comunicação entre os outros nós, semelhante a como opera um único sistema de CPU-GPU. A CPU é o *master* e serve como um *hub* de comunicação para grupos de *threads* de execução na GPU chamados blocos de *thread*.

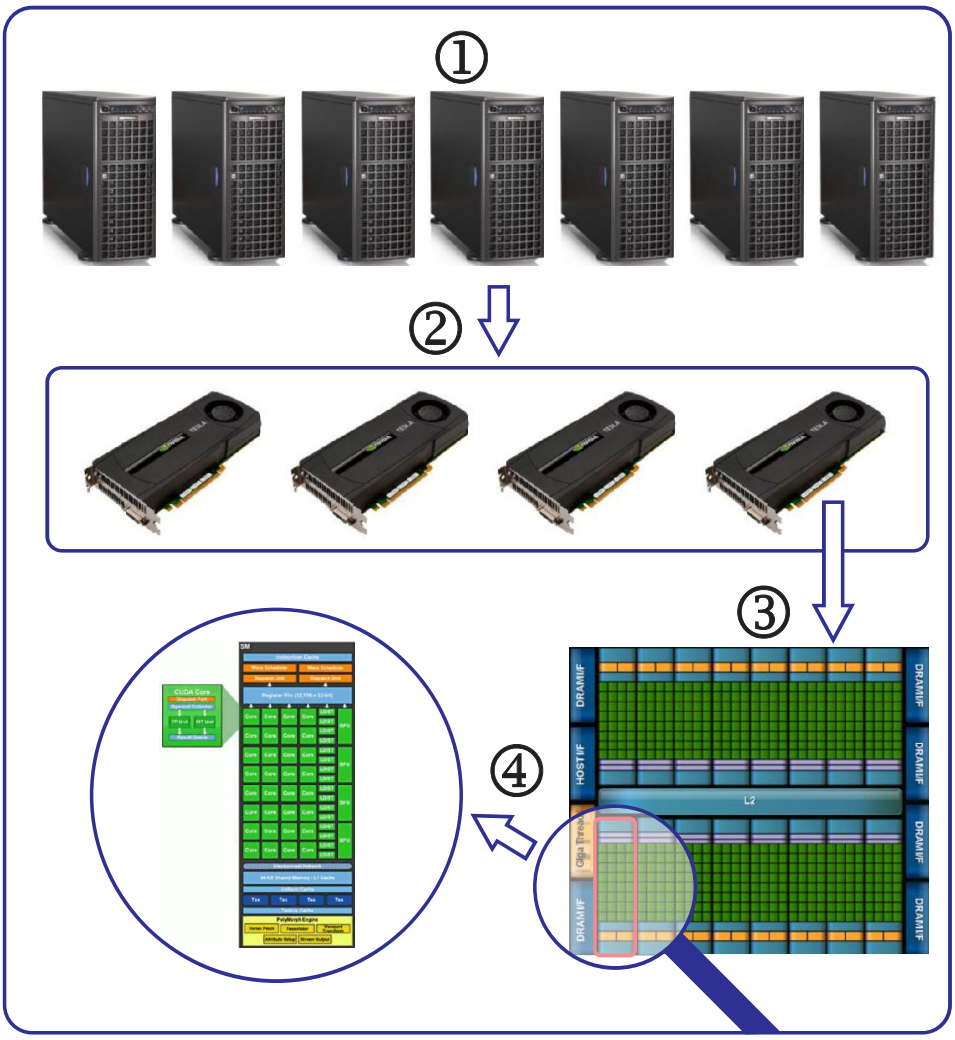


Figura 16. Vários níveis de paralelismo.

Cada bloco de *thread* é em si um conjunto de tópicos que podem se comunicar através de um espaço de memória apropriadamente chamado "memória compartilhada". Quando todos estes sistemas são utilizados em conjunto, a arquitetura resultante tem vários níveis de paralelismo. Os níveis de paralelismo de uma arquitetura multi-GPU são mostrados na Figura 16 na qual: (1) *Cluster* de um sistema comunicando-se através de uma LAN; (2) Múltiplas GPUs por sistema

comunicando-se através da DRAM do sistema; (3) Vários *Streaming Multiprocessor*s por GPU executando blocos de *thread* que se comunicam através da memória global da GPU; (4) Vários núcleos CUDA por multiprocessador executam *threads-warps* e se comunicam através da memória compartilhada.

Outra questão é que devem ser realizadas várias descomposições de domínio a fim de utilizar plenamente as capacidades deste sistema. A descomposição grosseira é muito semelhante à utilizada para os sistemas de MPI, mas a descomposição fina pode ser muito diferente, devido à largura de banda significativamente maior e menor da *cache* das GPUs.

2.4.6. Torque

O Torque, que é semelhante ao PBS (*Portable Batch System*), permite o melhor gerenciamento dos recursos disponíveis nos servidores do ICA/PUC-Rio.

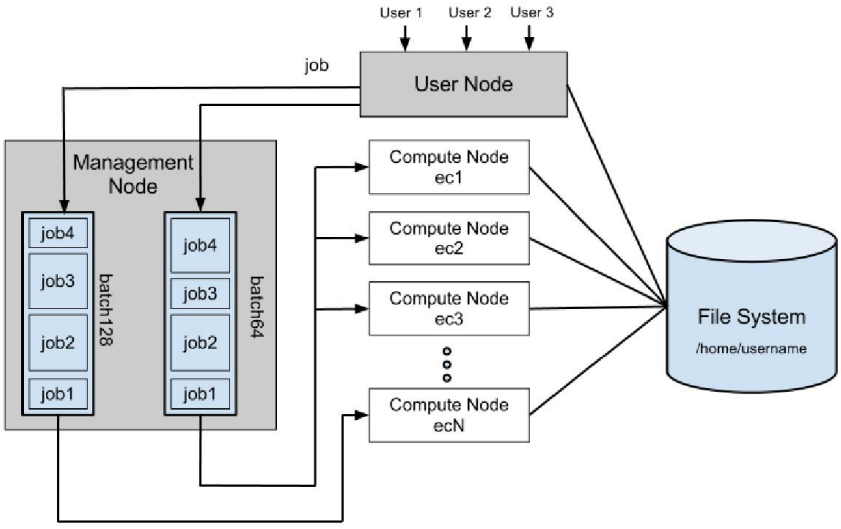


Figura 17. Gerenciador de Tarefas Torque

Por este motivo, nos servidores principais, a utilização dos recursos só será possível através do sistema de fila Torque. Para sua utilização o usuário indica para o Torque os recursos de que precisa para executar sua aplicação, e ele gerencia os recursos do *cluster*, também facilitando a integração dos nós para executar programas utilizando MPI. Um diagrama do funcionamento do Torque é apresentado na Figura 17.

2.5. Visualização Gráfica

2.5.1. Visualização de Dados Técnicos

Existem muitos métodos para visualizar informações. O método escolhido para um determinado conjunto de dados é frequentemente determinado pelo tipo de dado. Se os dados são estruturados com as conexões entre dados já conhecidos, então estas conexões devem ser aproveitadas, visualizando os dados e as ligações entre eles. Grande parte dos dados em PG são deste tipo e serão examinados, como pode ser visualizado na seção 3.3. Se os dados são desestruturados, então a finalidade da visualização é frequentemente encontrar as ligações entre eles.

Avaliação da eficácia de uma visualização muitas vezes não é fácil. Tufte [21] é uma das fontes mais confiáveis sobre o assunto. Ele estabeleceu uma série de diretrizes para a exibição de informações quantitativas. Estes incluem (1) mostrar os dados, (2) induzir o espectador a pensar sobre o conteúdo, em vez de como ele foi criado, (3) evitar a distorção dos dados, (4) apresentar muitos números em um espaço pequeno, (5) fazer grandes conjuntos de dados coerentes, (6) incentivar o olho a fazer comparações (7), revelar os dados em vários níveis de detalhe (8), servir a um propósito razoavelmente claro, e (9) estar estreitamente integrado com as descrições estatísticas e verbais. Vamos usar esses princípios para avaliar métodos de visualização posteriores.

2.5.2. Visualização em Computação Evolucionária

Tem sido observado em [22] que entender como a evolução age é um problema chave e que tem havido um número significativo de tentativas para visualizar os comportamentos. Eles classificam as visualizações como relação de informação, relacionando o *pedigree* das soluções, informações como respeito à convergência, ou informação sobre a paisagem de aptidão. Um conjunto de técnicas padronizadas para a visualização da computação evolutiva também foi definido em [23]. Estes incluem gráficos do tempo *versus* melhor aptidão e aptidão média, valores de aptidão de todos os indivíduos, e a distância entre indivíduos de uma

população. A visualização dos indivíduos em PG é mais complexa do que em outra representação em computação evolutiva.

2.5.3. Robótica evolutiva

Antes de serem reconhecidos como métodos de optimização eficazes, os primeiros algoritmos evolutivos resultaram de um desejo de transferir a riqueza e a eficiência dos organismos vivos aos artefatos e, em particular, para os agentes artificiais [24] [25] [26]. Esta visão futurista inspirou todo um campo de pesquisa em que os algoritmos evolutivos são empregados para automaticamente desenhar robôs ou controladores de robôs. Este campo é agora chamado Robótica Evolutiva (RE) [27] [28] [29] [30].

De acordo com pesquisadores de Robótica Evolutiva, os Algoritmos Evolutivos são candidatos promissores para enfrentar o desafio do desenho automático devido à sua capacidade de agir como "*blind watchmakers*" (em português, relojoeiros cegos): eles só se preocupam com o resultado sem restringir os princípios de funcionamento das soluções e, consequentemente, um algoritmo evolutivo poderia desenhar um robô baseado em propriedades incomuns que não se encaixam em os métodos de engenharia clássicos. Esse raciocínio é perfeitamente ilustrado pelo projeto Golem [31], em que Lipson e Pollack deixaram a estrutura de um robô e seu neuro-controlador evoluir, enquanto selecionavam os robôs por sua velocidade de translação. Para um processo evolutivo, estes autores obtiveram morfologias originais e andamentos simples mais eficientes. Em outro estudo típico, Doncieux e Meyer desenvolveram a rede neural que controlava um dirigível lenticular [32] e os Algoritmos Evolutivos encontraram um comportamento original que aproveita a forma do dirigível para criar sustentação adicional. Os Algoritmos Evolutivos são interessantes, não só porque eles levam a soluções eficientes, mas também porque essas soluções não são influenciadas por considerações humanas. Por conseguinte, elas são pelo menos inesperadas, se não inovadoras.

Os *benchmarks* comuns que têm sido estudados até o momento incluem locomoção simples, locomoção com a evasão de objeto [33] [34], fototaxia (movendo em direção a fontes de luz), e aprender a andar, no caso de robôs com pernas [35] [36]. Os primeiros trabalhos utilizaram as técnicas *Embodied Evolution*

(evolução em robôs fisicos) ou evolução em simulação com transferência para robôs reais após o processo evolutivo ter sido completado.

Na técnica *Embodied Evolution*, os robôs físicos são utilizados durante o processo evolutivo [37], [38], [39]. Basicamente são testados e as aptidões dos controladores associados são avaliadas com base ao desempenho dos robôs reais (Figura 18).

Figura 18. Técnica *Embodied Evolution*

Embora este procedimento acredite que os controladores possam funcionar em robôs reais (em oposição às simuladas), o processo é lento (em tempo real). Um problema adicional e mais sério é que mesmo aos piores controladores não podem ser permitidos danificar os robôs reais durante os testes, porque isso acabaria com o processo evolutivo, pelo menos até que os robôs pudessem ser reparados ou outro pudesse ser construído. O que isso realmente significa é que a técnica *Embodied Evolution* não pode fazer uso de medidas de *fitness* que medem a verdadeira capacidade de sobrevivência dos robôs. O programador deve decidir quais são os comportamentos que um robô provavelmente vai precisar para executar a tarefa, sem causar danos a ele. Este é um problema quando o objetivo é conseguir que os robôs aprendam como fazer algo que os programadores não sabem como fazer.

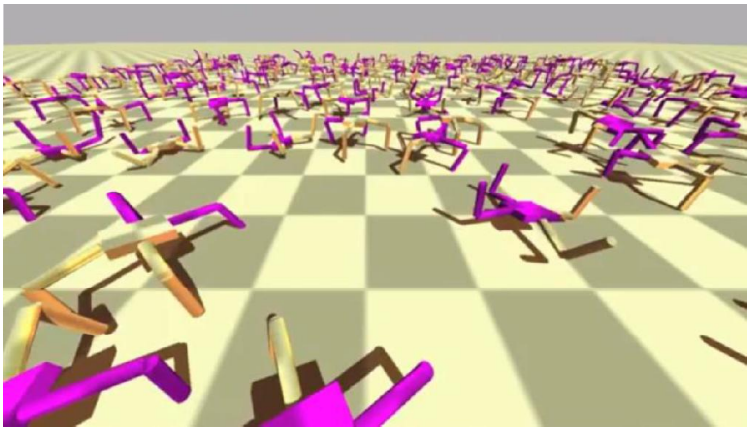


Figura 19. Ambiente Simulado

Uma alternativa para a técnica *Embodied Evolution* é evoluir os controladores em robôs simulados que vivem em ambientes simulados. Agora os robôs podem ser destruídos durante os testes e o condicionamento físico pode basear-se mais diretamente na sobrevivência real. No longo prazo, isso é muito importante. No entanto, devemos salientar que, para o estado atual da investigação sobre robótica evolutiva, os robôs geralmente simplesmente têm sucesso ou falham ao executar suas tarefas dadas e não enfrentam desafios mortais. A evolução na simulação pode avançar muito mais rápido do que a evolução usando apenas robôs reais. Tem-se que tomar cuidados na concepção dos ambientes de simulação para que os controladores evoluídos em simulação possam funcionar em robôs reais.