UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL GIRARDELLO DETONI

MTC: Modelo de Programação Paralela Baseado na Perspectiva Conexionista

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Dante Augusto Couto Barone
Orientador

Porto Alegre, outubro de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Detoni, Gabriel G.

MTC: Modelo de Programação Paralela Baseado na Perspectiva Conexionista / Gabriel G. Detoni – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

116 p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2010. Orientador: Dante Augusto Couto Barone.

1.Processamento Paralelo. 2.Multicore 3.Futebol de Robôs. I. Barone, Dante A. C..

Agradecimentos

Este trabalho é, antes de tudo, o fruto da inesgotável paciência, disponibilidade e conhecimento do meu orientador Prof. Dante Augusto Couto Barone, cujo suporte e incentivo foram inestimáveis, e a ele dedico meus mais sinceros agradecimentos.

Da mesma maneira, gostaria de registrar minha gratidão por minha família e à minha namorada Daniela, pela compreensão durante todo o percurso que, sob alguns pontos de vista, foi tão difícil para mim quanto foi para eles.

Pela flexibilidade, agradeço à minha empresa HP Computadores do Brasil, e em especial à minha equipe de trabalho: Julio, Wagner, João, Edison, Luis, Josi e Caren; pelas várias vezes em que meus compromissos com o mestrado os fizeram obrigados a assumir as minhas responsabilidades.

Por fim, agradeço ao professor Philippe Navaux e seu grupo de pesquisa (GPPD UFRGS) por disponibilizarem a infraestrutura necessária para realização de diversos testes durante o percurso deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

TC	Tarefa Conexionista
MTC	Modelo de Tarefas Conexionistas
PDP	Processamento Paralelo Distribuído
SMT	*Simultaneous Multithreading*
NUMA	Acesso não uniforme à memória
UE	Unidade de Execução
UP	Unidade de Processamento
UC	Unidade de Controle
SSL	*Small Size League*
MSL	*Medium Size League*
IA	Inteligência Artificial
MIMD	*Multiple Instruction Multiple Data*
SIMD	*Single Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
SISD	*Single Instruction Single Data*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Neste trabalho é apresentado o desenvolvimento de um modelo de programação paralela inspirado na estrutura geral dos sistemas conexionistas como proposta por Rumelhart, McClelland e Hinton em seu livro intitulado *The Parallel Distributed Processing Perspective* (MCCLELLAND, RUMELHART e HINTON, 1983). Este modelo tem como objetivo servir como base para o desenvolvimento de um sistema autônomo de controle em tempo hábil para um time de futebol de robôs, orientado ao uso de processamento paralelo em computadores *multicore*. O trabalho serve como um guia para compreensão e uso do modelo de programação proposto, apresentando ainda, por meio de experimentos, a sua eficiência dentro do contexto de processamento paralelo e a sua adequação para solução do problema de controle de futebol de robôs.

Palavras-Chave: Processamento Paralelo, conexionista, PDP, multicore, SSL, Robocup.

MTC: Parallel Programming Model based on the Connectionist Perspective

ABSTRACT

This work presents the development of a parallel programming model inspired by the general framework of connectionist systems as proposed by Rumelhart, McClelland and Hinton in their book entitled *The Parallel Distributed Processing Perspective* (MCCLELLAND, RUMELHART e HINTON, 1983). This model is intended to serve as the basis for the development of an autonomous system for real-time control of a robotic soccer team driven towards the usage of effective parallel processing on multicore computers. The work serves as a guide for understanding and using the proposed programming model, yet showing, through experiments, the efficiency within the context of parallel processing and its suitability for solving the robotic soccer control problem.

Keywords: Parallel Processing, connectionist, PDP, multicore, SSL, Robocup**.**

1 INTRODUÇÃO

"Um movimento irreversível em direção aos processadores multicore está em andamento. Construir processadores multicore garante a promessa da Lei de Moore, mas cria um enorme problema para os desenvolvedores. Processadores multicore são computadores paralelos, e computadores paralelos são notoriamente difíceis de programar"

- Charles E. Leiserson (LEISERSON e MYRMAN, 2008)

Esta frase oferece uma idéia da força do movimento industrial em direção aos computadores paralelos. Ela também coloca em pauta o desafio imposto aos desenvolvedores neste campo, onde há necessidade de mecanismos e técnicas que facilitem a programação, já que existe uma série de problemas característicos da execução paralela que não está presente na seqüencial, e é apenas nesta última, que grande parte dos programadores de hoje em dia possui experiência.

O presente trabalho apresenta o desenvolvimento de um modelo de programação inspirado na teoria do Processamento Paralelo Distribuído (PDP) com objetivo de servir como ferramenta para o desenvolvimento de aplicações de controle em tempo hábil utilizando processamento paralelo, orientado a computadores *multicore*. Esta teoria, com mais de duas décadas de idade, contém, ainda hoje, a descrição analítica mais utilizada dentro da ciência da computação para definir um sistema inteligente biológico. Por utilizar a inspiração em sistemas biológicos de natureza assíncrona, esta é a teoria escolhida para fundamentar este trabalho, uma vez que estes são altamente eficazes na solução de problemas de tempo hábil e igualmente eficazes na utilização de paralelismo.

Alguns motivos levaram à escolha deste objetivo. Em primeiro lugar, modelar soluções e as desenvolver, de maneira orientada ao processamento paralelo, constitui-se em uma tarefa difícil, mas o aumento no número de núcleos dos processadores comerciais torna necessário software adequado a esta evolução. Ignorá-la implica no alto risco de tornar o software incapaz de utilizar máquinas mais potentes de forma plena, reduzindo, desta forma, a sua vida útil.

Por fim, este é um trabalho multidisciplinar, e seu campo de interesse é a aplicação de processamento paralelo ao controle de robôs móveis por meio de um único computador central. Esta é a plataforma padrão de uma liga específica de futebol de robôs conhecida como *Small Size League* (SSL) onde se deseja realizar uma contribuição prática por meio do desenvolvimento de um sistema de controle baseado no modelo de programação proposto.

Para a plataforma da SSL, existem duas grandes classes de sistemas de controle. A primeira é inspirada na biologia, e consiste em considerar cada robô como uma

entidade independente que controla a si própria e produz comportamento coletivo através do uso de comunicação implícita ou explicita para coordenação dos agentes. Este tipo de sistema, também chamado multi-agentes, apresenta uma série de características que o tornam a abordagem ideal para problemas onde os agentes são homogêneos e numerosos, possuem metas simples, e não exigem comunicação intensa. Um forte exemplo deste tipo de problema é aquele que pode ser resolvido por um enxame ou colônia de insetos. Como cada robô desempenha suas atividades de maneira independente e simultânea, este tipo de solução se enquadra no campo de robótica móvel distribuída e os seus desafios de um ponto de vista de processamento paralelo são bastante conhecidos e satisfatoriamente contornados.

A segunda classe utiliza um único agente autônomo que controla todos os jogadores. Em uma analogia, este agente movimenta seus jogadores de forma semelhante a uma pessoa que observa um jogo de xadrez e movimenta as suas peças de acordo com suas observações. Nesta classe é utilizado um sistema de visão e atuação absoluto, que permite ao agente enxergar e manipular o jogo como um todo (e não relativo a cada robô). Embora esta categoria possua maiores restrições, ela é a melhor sucedida. Por considerar um único agente, obter vantagem de computadores paralelos passa a ser um desafio complexo e ao mesmo tempo, oferece um potencial alto para a obtenção de vantagens competitivas.

Nos últimos cinco anos (de 2006 a 2010), apenas quatro times participaram das finais da Robocup na liga SSL, quase todos eles se classificando em primeiro ou segundo lugar por mais de uma vez. São eles Skuba, Plasma-Z, RoboDragons e CMDragons. Todos estes times se enquadram na segunda classe, de forma que ela comprovadamente apresenta melhor desempenho neste tipo de competição.

No entanto, nenhum destes times, utiliza algoritmos baseados em *multi-threading* para o controle do seu time. Em (ZICKLER, BRUCE, *et al.*, 2009) é citado como razão, o fato de futebol de robôs se enquadrar em um ponto intermediário entre o que seria um domínio altamente paralelo, onde robôs completam partes da tarefa individualmente, e um domínio serial, onde cada parte da tarefa é feita por um robô por vez. Isto se deve em especial à presença de apenas uma bola, de forma que apenas um robô atua ativamente sobre ela e os demais apenas o acompanham provendo suporte. O presente trabalho leva este fator em consideração, de forma que procura dividir cada atividade realizada por um agente em várias operações e assim paralelizá-las, ao invés de simplesmente dividir a grande tarefa a ser realizada pelo sistema entre vários agentes; o que em essência o colocaria na primeira classe (*multi-agentes*).

Por fim, um sistema multi-agentes é geralmente associado à boa escalabilidade, intrínseca da possibilidade de se remover ou adicionar agentes à medida que a aplicação requer ou suporta, a esta característica dá se o nome fluidez (KIRKWOOD-WATTS, 2000). Em um time de futebol de robôs há um número fixo e pré-definido de robôs que permite pouca fluidez à solução, tornando um sistema multi-agentes pouco escalável de um ponto de vista de processamento paralelo. Mesmo com a adição de novos processadores ao computador que executa o sistema, não será possível adicionar novos agentes para utilizá-los.

Em vista destes fatores, a solução proposta neste trabalho se enquadra na segunda classe, constituindo um sistema com um único agente, e o motivo que leva à procura de um modelo de programação adequado é o fato de esta classe impor desafios à aplicação de processamento paralelo, já que, ao contrário de um sistema multi-agentes, ela não é intuitivamente divisível e, portanto paralelizável.

A opção tomada neste trabalho foi pelo desenvolvimento de um sistema novo ao invés de explorar o paralelismo em um sistema já existente. O motivo desta escolha, como será apresentado mais adiante, é o fato de o potencial para ganho de desempenho em um sistema projetado desde o principio para execução em paralelo ser muito maior do que o de um sistema seqüencial que é adaptado ao paralelismo.

O objetivo geral deste trabalho é, portanto, desenvolver um modelo de programação paralelo baseado no conexionismo e aplicá-lo ao futebol de robôs. Os objetivos específicos são: apresentar o campo de futebol de robôs; compilar fundamentos do processamento paralelo para situar o modelo desenvolvido dentro dessa área; e por fim, apresentar o modelo desenvolvido e sua aplicação ao futebol de robôs.

O texto está estruturado da seguinte maneira: o capítulo 2 apresenta a área de futebol de robôs situando a contribuição do trabalho a uma de suas principais ligas; o capítulo 3 apresenta a área de processamento paralelo e o capítulo 4 detalha os conceitos e jargões da programação paralela. No capítulo 5 é mostrado o modelo de programação desenvolvido. No capítulo 6 é feita a aplicação deste modelo ao futebol de robôs produzindo um sistema de controle para futebol de robôs. O capítulo 7 apresenta os testes realizados com o modelo de programação bem como com o sistema de controle para futebol de robôs desenvolvido. Por fim, as conclusões obtidas são descritas no capítulo 8.

2 FUTEBOL DE ROBÔS

A robótica é um ramo da tecnologia que envolve mecânica, eletrônica e computação de maneira a produzir máquinas mais versáteis do que seria possível a partir de cada um destes ramos individualmente. A estas máquinas dá se o nome de robô.

A definição de robô, no entanto, é bastante controversa já que cada uma das áreas envolvidas coloca suas próprias variantes. Por exemplo, existem máquinas mecânicas, autônomas porém analógicas e não eletrônicas. Existem programas para computadores eletrônicos que são autônomos, não exigindo interação de um humano para que produzam seus resultados, no entanto tais resultados não produzem alterações no meio físico, não contendo a parte mecânica. Este tipo de programa é também conhecido como Robô de Software ou *Bot*. Por fim, existem máquinas mecatrônicas, controladas remotamente por humanos e não auto-geridas por um sistema computacional.

Tentando uniformizar o conceito de robô, a International Organization for Standardization (ISO) define um robô como "um manipulador automaticamente controlado, re-programável, de múltiplos propósitos e possuindo três ou mais eixos." (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1994). Esta é, portanto, a definição de robô utilizada neste trabalho.

A partir da metade do século XX, com a chegada dos microchips que tornaram a eletrônica barata, a robótica passou a ser vista como uma alternativa viável e desde então, vem ganhando força com crescente interesse da indústria em automatizar atividades complexas do processo de fabricação de bens de consumo de forma a aumentar a produtividade.

Este crescimento da robótica industrial alavancou o desenvolvimento de componentes mecânicos como motores, atuadores, amortecedores entre outros. Juntamente à mecânica, a computação vem propondo melhorias constantes nos mecanismos para o controle autônomo de robôs.

O futebol de robôs é uma variedade relativamente recente da robótica, criada em 1993 (Robocup, 2010). Esta área procura criar um campo de provas comum para o estudo e integração de uma vasta gama de tecnologias. Ao colocar diferentes técnicas em conflito, o futebol de robôs consegue ao mesmo tempo classificar o seu desempenho e incentivar o seu desenvolvimento através da competição.

2.1 Robocup

Duas entidades promovem eventos de futebol de robôs em escala mundial, a FIRA e a Robocup, sendo a segunda a escolhida para estudo neste trabalho. A sua missão é "Até ao ano 2050, desenvolver uma equipe de robôs humanóides totalmente autônomos capazes de derrotar a equipe campeã mundial de futebol humana." (Robocup, 2010).

A Robocup foi fundada em 1997 com o fim de conduzir competições mundiais entre robôs, sendo o futebol de robôs a sua principal modalidade. Esta organização tem crescido de forma constante, com a participação de cada vez mais competidores em todo o mundo.

Inicialmente, o futebol foi a grande motivação para o Robocup. O futebol é um esporte bastante popular em escala mundial, mas apresenta também desafios científicos bastante importantes, isto porque é um jogo com problemas individuais como movimentação e sensoriamento, e ao mesmo tempo, problemas coletivos como estratégia e jogadas. A existência deste ambiente dinâmico e competitivo requer times sofisticados para que sejam capazes de atuar de maneira eficaz.

Na Robocup existe uma série de ligas, cada uma focando em diferentes aspectos da tecnologia envolvida no futebol de robôs, como por exemplo, a visão computacional, a comunicação entre agentes e o estudo da inteligência artificial.

Para cada liga são passadas, aos competidores, especificações detalhadas de todo o ambiente de competição como tamanho, peso, formato dos robôs, medidas do campo de futebol e regras a ser seguidas durante os jogos. A seguir estas ligas são apresentadas.

2.1.1 Liga Simulada - *Simulation League*

Nesta liga não existem robôs físicos, todo o ambiente e os agentes são simulados. É fornecido a todos os competidores um mesmo simulador que provê um ambiente completo para um jogo de futebol de robôs. Aos competidores cabe desenvolver o sistema de controle que fará com que estes robôs atuem em equipe. O simulador fornece aos agentes todos os dados que seriam obtidos na realidade, através dos seus sensores, e calcula o resultado das ações de cada agente. Cada agente é um processo separado e existem, no total, 11 agentes em cada equipe. Existem duas modalidades nesta liga: simulação 2D e 3D.

2.1.2 Liga de Quadrúpedes - *Four-legged League*

Esta liga é jogada por robôs de 4 patas. São equipes que utilizam 5 robôs AIBO da Sony. Estes robôs têm os sensores necessários incorporados e comunicam com os restantes elementos via wireless provendo toda a plataforma de hardware. Cabe aos times desenvolverem o sistema de controle.

Figura 2.1: Cão robótico AIBO da Sony utilizado nas competições da Four-legged league.

2.1.3 Liga Humanóide - *Humanoid League*

Esta competição utiliza robôs humanóides, portanto mais complexos. Desta forma, esta liga foca ainda em aspectos básicos para jogar futebol como movimentar-se no campo, controlar a bola e não nos aspectos coletivos. Por este motivo, nesta liga ainda não há jogos de futebol como nas demais. É a liga mais atrativa mas igualmente aquela que apresenta os maiores desafios e de maior complexidade. A intervenção humana nesta liga é permitida, já que em muitos casos, os robôs não possuem um sistema de controle autônomo.

Figura 2.2: Exemplos de robôs humanóides utilizados na Robocup 2006.

2.1.4 E-league

Esta é a mais recente liga do Robocup, aparecendo pela primeira vez no Robocup de 2004 em Lisboa. Foi criada para permitir competições entre equipes de baixo custo, servindo como treinamento preparatório para equipes que desejam entrar nas outras modalidades no futuro. Centra-se nos problemas de alto-nível, usando plataformas simples, usando tecnologias generalizadas quanto à visão e comunicação,

2.1.5 Liga de Tamanho Médio - *Middle Size League (MSL)*

Jogada por duas equipes de 5 robôs móveis autônomos movidos por rodas, com uma altura que varia entre os 50cm e os 90cm. Toda a informação relativa ao ambiente é obtida pelos seus sensores e são capazes de comunicar-se via wireless. Tirando a introdução e remoção de robôs do terreno de jogo e recolocação da bola em campo, não é permitida qualquer intervenção humana durante o jogo.

Figura 2.3: Fotografia de um jogo da MSL.

2.1.6 Liga de Tamanho Pequeno - *Small Size League (SSL)*

Nesta liga os robôs são bem menores e o ambiente é muito mais controlado para lhes facilitar a vida. Embora sejam independentes uns dos outros, todo o processamento é efetuado num computador central, baseado em informação vinda de uma câmara de vídeo colocada acima do campo, sendo os comandos enviados para os robôs via wireless. A intervenção humana resume-se à introdução e remoção dos robôs no campo.

Figura 2.4: Fotografia de um jogo da SSL.

2.2 Estudo da SSL

Exceto pela liga humanóide, uma característica comum a todas estas ligas é a necessidade de um sistema de controle autônomo, porém, as diferentes características de cada liga limitam ou forçam a escolha do modelo de software a ser usado. Para a MSL, onde os robôs possuem um sistema de visão individual e menos restrições de peso e tamanho, é comum que se utilize sistema multi-agentes e cada robô tome suas decisões individualmente, se comunicando com os demais por rádio. A SSL pelo contrário é bastante restritiva quanto ao tamanho dos robôs, e estabelece um sistema de visão global, por meio de uma câmera montada acima do campo e o time inteiro é controlado por meio de um computador central.

A liga escolhida para estudo neste trabalho é a SSL em função do seu baixo custo operacional comparada às outras ligas que utilizam robôs de maior porte, sendo geralmente a liga escolhida como ponto de partida pelas instituições que desejam participar em competições.

O grupo do Programa de Educação Tutorial (PET) em Computação da UFRGS financiado pela Secretaria de Educação Superior do MEC possui o projeto RoboPET que conta com a infra-estrutura necessária para participar da SSL. Esta equipe já participou das competições Robocup 2004 em Lisboa e Robocup 2009 em Graz, na Áustria. Por este motivo, as contribuições desenvolvidas por este trabalho têm como objetivo a utilização prática.

2.2.1 Estado da Arte

De um ponto de vista de hardware, embora existam várias formas de desenvolver um time para esta liga, o mecanismo mais utilizado hoje pelas equipes consiste de sete robôs homogêneos sendo que apenas cinco deles participam do jogo a cada momento. Dois servem apenas como reserva em caso de falha de algum dos jogadores.

Os robôs são omnidirecionais e movidos por três ou quatro motores elétricos, cada um possui um chutador movido por um atuador elétrico de força variável para efetuar passes e chutes, além de um driblador movido por um motor elétrico com o objetivo de conduzir a bola sob o controle do robô.

O time Plasma-Z da Chulalongkorn University (KRIENGWATTANAKUL, 2008), campeão da Robocup SSL em 2008 serve para exemplificar a tecnologia utilizada atualmente nas competições. Nele há quatro sistemas distintos, o sistema mecânico, o sistema elétrico, o sistema de visão e o sistema de Inteligência Artificial (IA).

O sistema mecânico dos robôs possui quatro motores de 30 W, atingindo uma velocidade máxima de 2,49 m/s e uma aceleração máxima de 11,62 m/s². Os mecanismos de chute são dois, um realiza chutes rasteiros, o outro chuta a bola para o alto, podendo deslocá-la a 60 cm de altura e 1 m de distância.

O sistema elétrico serve como suporte para o firmware dos robôs e é implementado sobre FPGA contendo componentes auxiliares como o sistema rádio para comunicação com o controle central utilizando a faixa de 900 MHz. Os pacotes de comunicação são recebidos do sistema de IA a uma freqüência de 60 Hz, sendo decodificados pelo controle central do robô para gerar sinais aos motores e atuadores.

O sistema de visão utiliza duas câmeras de 640x480 pixels, cada uma cobrindo uma metade da quadra. As câmeras se conectam a um computador central por uma placa PCI a uma taxa de transmissão de 400 Mb/s, sendo capazes de processar 60 quadros por segundo. Para que o sistema se comporte adequadamente é necessário calibrar o sistema de visão antes dos jogos.

A localização de objetos pelo sistema de visão é uma tarefa bastante complexa, em especial a bola devido ao seu tamanho e velocidade. No Plasma-Z, uma análise

preliminar é feita via separação de cores definindo alguns elementos candidatos, após, análises heurísticas detectam os objetos considerando diferentes possibilidades.

O sistema de IA é um processo executado em um computador dedicado que trata dados recebidos do sistema de visão e os converte em comandos a cada robô, transmitidos via rádio.

Como ponto inicial do sistema de IA, filtros de Kalman (WELCH e BISHOP, 2006) são utilizados para estimar e prever a posição dos objetos, diminuindo a sensibilidade a ruídos provenientes do sistema de visão. Outra função destes filtros é estimar a posição real dos robôs e da bola já que existe um pequeno intervalo entre os dados processados pelo sistema de visão e a situação verdadeira no campo.

O principal componente da IA é o código que define a estratégia, planeja e coordena as ações dos robôs. Para cada time o sistema de IA é diferente, porém aqueles com melhores resultados possuem um processamento o mais simples possível, dentro de um laço com iterações curtas. Ainda, como parte da IA, o Plasma-Z possui um sistema de aprendizado baseado em Redes Neurais Artificiais que tenta ajustar em tempo hábil parâmetros de correção dos movimentos dos robôs. Outros times como o ITAM's Eagle Knights utilizam um sistema similar, porém, neste caso, este sistema serve para ajustar o sistema de visão (TORRES e WEITZENFELD, 2008).

Embora na grande maioria das situações o sistema de controle autônomo do time de robôs empregue técnicas de IA isto não é uma exigência, e há situações em que isto não ocorre. Por este motivo, no restante do documento, em lugar do termo IA, será utilizado o termo "sistema de controle autônomo".

2.2.2 Pontos a Evoluir

A Robocup tem se demonstrado um excelente campo de provas para tecnologias em robótica ao longo dos anos, e ainda há muito para evoluir até que se obtenham times de robôs que se comportem de maneira inteligente.

No que diz respeito às áreas sujeitas a melhorias, os sistemas mecânico e elétrico são limitados propositadamente pelas regras de cada liga, porém ambos possuem pontos a aprimorar já que a evolução nestas áreas, de um modo geral, é constante e sempre se pode tirar proveito de progressos nos materiais, fontes de energia e componentes elétricos e eletrônicos. As tecnologias disponíveis para a visão também podem ser melhoradas com a disponibilidade de câmeras de vídeo mais poderosas e softwares mais eficientes para processamento visual. Por fim, o sistema de controle autônomo existente hoje nas equipes vencedoras é bastante eficaz em realizar a coordenação dos jogadores, porém ainda requer muitas melhorias para que possa ser considerado verdadeiramente inteligente. Por esta razão, esta área tem sido o foco das principais equipes da atualidade como tendo o maior potencial para avanços, já que o investimento necessário para sua evolução é inferior às demais, consistindo basicamente em pesquisa e desenvolvimento de software.

Este trabalho leva em conta estas premissas estabelecendo seu foco na contribuição para a área de software, mais especificamente no sistema de controle autônomo do time de robôs.

Como mencionado anteriormente, o estado da arte em relação ao sistema de controle autônomo, no que diz respeito à estratégia, planejamento e coordenação dos jogadores é baseado em um processo iterativo que utiliza um laço onde o processamento é realizado em série e por fim transmitido aos robôs como comandos.

Acompanhando a tendência da indústria de computadores em oferecer processadores com um número cada vez maior de núcleos (*multicore*), hoje é possível e desejável que softwares sejam desenvolvidos sem as limitações do processamento em série, mas para atingir este objetivo é necessário que os programas sejam projetados e construídos levando em conta a execução de código em paralelo. Portanto, neste trabalho será estudado o desenvolvimento de um sistema de controle autônomo para um time de futebol de robôs o qual obtenha benefício do uso deste tipo de processamento.

Algumas hipóteses bastante aceitas, como a mencionada em (MORAVEC, 1998), consideram que os computadores atuais não possuem a capacidade de processamento e armazenamento necessários para o desenvolvimento de uma inteligência artificial forte (equivalente à humana), e, portanto, para o desenvolvimento de sistemas de controle autônomos altamente inteligentes. O uso de processamento paralelo permite que mais informação seja tratada em menos tempo, e isto é sem dúvida um avanço no aumento da capacidade de processamento, em relação a sistemas que utilizam processamento em série. Por isto, embora seja possível construir times eficazes para futebol de robôs com computadores baseados em um único processador, a utilização de processamento paralelo deve ser entendida como um meio para que os sistemas de controle autônomos tenham a sua disposição uma capacidade de processamento cada vez maior, e com isto possam se aproximar cada vez mais de um sistema realmente inteligente.

Dentro da área de Inteligência Artificial, é possível classificar o sistema de controle de um time de futebol de robôs, como o proposto neste trabalho, como sendo um agente reativo baseado em modelo, pois reage ao seu ambiente com base em uma representação interna do mundo construída a partir de suas percepções. O agente é ainda, racional, pois toma decisões procurando aumentar sua medida de sucesso, e autônomo por fundamentar suas ações em suas percepções.

O ambiente em que este agente atua é estratégico, pois é determinístico exceto pelas ações de outros agentes. Este ambiente é seqüencial e dinâmico, pois as ações dos agentes possuem efeitos de longo prazo e o ambiente pode alterar-se enquanto um agente está deliberando. É ainda, contínuo e parcialmente observável, pois possui um número infinito de estados possíveis e não é possível ao agente definir completamente um estado a partir de suas observações. Por fim, é um ambiente multi-agente devido à presença de um oponente. O conjunto destas características torna o desenvolvimento deste tipo de sistema bastante desafiador.

3 PROCESSAMENTO PARALELO

Neste capítulo será apresentada uma visão geral do processamento paralelo como área da Ciência da Computação. A seguir, são elencados alguns conceitos chave para entendimento deste trabalho, focando no hardware e software necessários para o processamento paralelo.

Processamento Paralelo consiste em atuar sobre dados simultaneamente. Tomado no contexto da Ciência da Computação, como mencionado em (HENNESSY e PATTERSON, 2000), consiste em "um único programa executando simultaneamente em vários processadores".

Processar informações de forma paralela abre o leque de alternativas para pesquisa e desenvolvimento de computadores mais velozes e, portanto, mais eficazes, o que, de outra forma, seria possível apenas através do aumento da freqüência de operação; técnica que, para as arquiteturas baseadas em silício, já está alcançando o seu limite teórico (TANENBAUM, 2003). Esta área ganhou interesse da indústria onde o desenvolvimento de tecnologias paralelas nesta ultima década está finalmente tirando proveito das teorias criadas durante os anos 80 e 90. Hoje já existem, em grande quantidade, processadores com múltiplos núcleos (multicore) e máquinas superescalares, sendo utilizados em diversas atividades do dia a dia, como consultas na internet, processamento gráfico em jogos e simulações, entre tantas outras.

Computadores paralelos, no entanto, são apenas uma parte da organização que resultará enfim no aumento do desempenho de uma atividade computacional. De maneira análoga ao hardware que é capaz de desempenhar atividades simultaneamente, existe a necessidade de software apto a executar atividades distintas ao mesmo tempo. Ainda referente a isto, para a grande maioria dos casos, existe uma relação bastante forte entre a arquitetura dos computadores paralelos e os softwares projetados e desenvolvidos para ser executado nos mesmos (KLAIBER e LEVY, 1994), de forma que aumentar o desempenho de uma determinada atividade computacional é um exercício de ajuste entre hardware e software que deve ser considerado desde a concepção do sistema.

O Processamento Paralelo é uma área bastante ampla, englobando teorias computacionais, aplicações em arquiteturas de computadores, sistemas operacionais e modelos de programação. Embora este trabalho não ignore estas áreas, o seu foco é especialmente direcionado ao estudo de modelos de programação aplicáveis a sistemas computacionais pessoais, onde situa sua contribuição.

Este capítulo tem por objetivo contextualizar o leitor nos principais tipos de hardware e software paralelos bem como nos conceitos e jargões utilizados na área.

3.1 Hardware Paralelo

O funcionamento de um computador paralelo depende, em grande parte, das arquiteturas de hardware utilizadas. A seguir são apresentadas duas classificações bastante utilizadas para arquitetura de computadores paralelos e, na seqüência, o estado da arte e tendências futuras para estas arquiteturas.

3.1.1 Classificação de Flynn

Há uma série de taxonomias de hardware de um ponto de vista de paralelismo. Embora bastante antiga, a classificação mais aceita na atualidade é a proposta por Flynn (FLYNN, 1972). Nesta classificação, o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas

Flynn propôs quatro classes de computadores que dependem da quantidade de fluxos de instruções e de dados, combinando as possibilidades:

- *Single Instruction Single Data Stream* (SISD): Fluxo único de instruções/Fluxo único de dados. Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados, não caracterizando uma maquina paralela.
- *Single Instruction Stream/Multiple Data Stream* (SIMD): Fluxo único de instruções/Fluxo múltiplo de dados. Envolve múltiplos processadores (escravos) sob o controle de uma unidade de controle (mestre), executando simultaneamente a mesma instrução em diversos conjuntos de dados. A figura 3.1 apresenta o diagrama de uma máquina SIMD, onde UC é a unidade de controle, UP são as unidades de processamento e M são as memórias. FI é o fluxo de instruções e FD o fluxo de dados. Exemplos de utilização são: manipulação de matrizes e processamento de imagens.

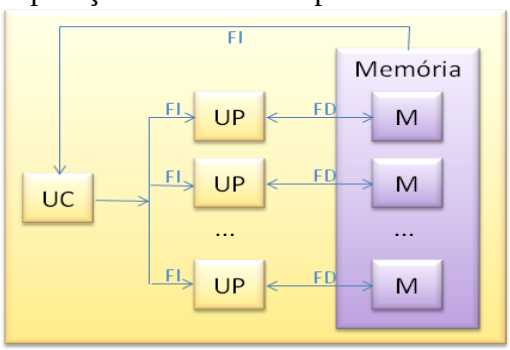


Figura 3.1: Diagrama de uma máquina SIMD.

- *Multiple Instruction Stream/Single Data Stream* (MISD): Fluxo múltiplo de instruções/Fluxo único de dados. Envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Embora haja discussões quanto à viabilidade prática desta arquitetura, boa parte dos estudiosos a considera vazia.

- *Multiple Instruction Stream/Multiple Data Stream* (MIMD): Fluxo múltiplo de instruções/Fluxo múltiplo de dados. Envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente. Esta classe engloba a maioria dos computadores paralelos. A figura 3.2 mostra o diagrama de uma máquina MIMD onde UP são as unidades de processamento e M são as memórias. FI é o fluxo de instruções e FD o fluxo de dados.

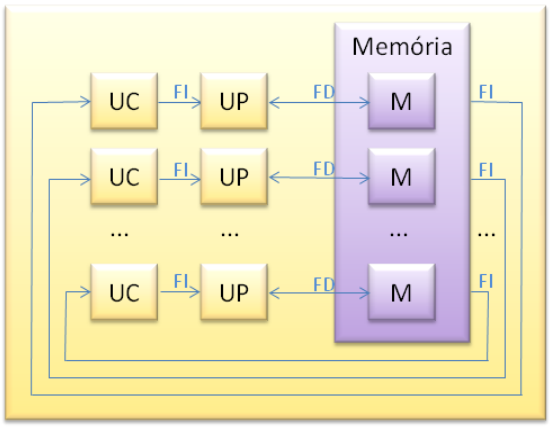


Figura 3.2: Diagrama de uma máquina MIMD.

É importante ressaltar que a classificação de Flynn desconsidera muitos avanços nas técnicas empregadas para obtenção de paralelismo, detendo-se apenas a instruções de processador. É possível, por exemplo, ampliar esta taxonomia para considerar granularidades maiores de computação, como o fluxo de controle e de dados de um programa inteiro, os quais, de forma similar às instruções, podem ser simples ou multiplos, originando outros tipos de máquinas como *Multiple Program stream/Multiple Data stream* (MPMD), *Multiple Program stream/Single Data stream* (MPSD) e *Single Program stream/Multiple Data stream* (SPMD) entre outras. Estas classes serão vistas em mais detalhe na seção 3.2.

Embora existam categorizações mais amplas, não há uma única classificação que considere todas as combinações de técnicas em uso na atualidade, pois estas são muito variadas e, em sua maioria, atuam em escalas diferentes ao mesmo tempo, se enquadrando, muitas vezes, em mais de uma categoria.

3.1.2 Classificação quanto ao Modelo de Comunicação

A comunicação entre tarefas está entre os maiores problemas a ser resolvido para que haja eficiência no uso do paralelismo, devido à relação entre desempenho e custo que ela impõe. De forma geral, quanto mais distante fisicamente duas unidades de processamento estão, mais lenta será a comunicação entre elas, e, por outro lado, quanto menor a distância, mais cara é a solução, implicando em menor escalabilidade e tolerância a falhas.

Uma vez que não existe solução ideal, é preciso escolher qual das características será beneficiada e desenvolver a arquitetura de hardware em função de sua aplicação. Algumas arquiteturas beneficiam o desempenho, outras a escalabilidade e custo. Devido a esta heterogeneidade, é bastante difícil desenvolver aplicações

cuja comunicação entre tarefas se comporte adequadamente em qualquer tipo de máquina paralela.

Algumas propostas de comunicação se destacam no que diz respeito ao desempenho e viabilidade prática (HENNESSY e PATTERSON, 2000). São elas as máquinas que utilizam memória compartilhada, e as que utilizam troca de mensagens. Estas ainda se dividem em máquinas com memória centralizada ou distribuída.

3.1.3 Multiprocessadores com Memória Compartilhada

Memória compartilhada consiste em compartilhar o espaço de endereçamento de dados, esta técnica está requer uma conexão física por meio de barramento e o acesso à memória é feito via primitivas *load/store*. Existem dois tipos básicos de multiprocessadores que compartilham o espaço de endereçamento, os *Uniform Memory Access* (UMA), também chamados *Symetric Multiprocessors* (SMP) onde todos os processadores possuem o mesmo tempo de acesso a qualquer posição de memória e as máquinas *Non-uniform Memory Access* (NUMA), onde os processadores possuem diferentes tempos de acesso dependendo de qual processador solicitou o acesso e a qual posição o acesso é feito.

Máquinas UMA possuem limitações quanto ao número de processadores que podem ser suportados, pois enfrentam problemas de desempenho em caso de compartilhamento do barramento, ou de custo no caso de possuir barramentos dedicados. Estas máquinas, em geral, não passam de uma dezena de processadores, já que a partir desta quantidade, torna-se inviável replicar barramentos para cada processador e o seu compartilhamento satura o canal de comunicação degradando o desempenho.

Máquinas NUMA são mais baratas e eliminam as limitações das máquinas UMA podendo chegar a centenas de processadores. Nestas máquinas, embora o espaço de endereçamento seja coletivo, diferentes partes da memória são acessadas mais rapidamente pelos processadores de acordo com a sua proximidade a estes.

Em função do espaço de endereçamento ser compartilhado, nestas arquiteturas a comunicação entre tarefas é implícita, não havendo diferença no acesso aos dados utilizados por uma tarefa apenas e aqueles alterados por mais de uma tarefa (caracterizando uma comunicação).

3.1.4 Multiprocessadores Baseados em Troca de Mensagens

Troca de mensagens consiste em definir espaços de endereçamento privados e compartilhar dados entre estes por meio de uma rede através de primitivas *send/receive*. Este tipo de sistema elimina a necessidade de modificações estruturais nos processadores e permite paralelismo em uma escala muito grande (centenas de processadores). Porém, a velocidade da comunicação entre os processos pode sofrer um grande impacto no desempenho já que fica limitada à velocidade da conexão de rede que costuma ser inferior à velocidade de um barramento de memória. Além disto, pacotes de rede introduzem sobrecarga (*overhead*) devido ao protocolo de comunicação utilizado, sendo naturalmente mais lento que a memória compartilhada. A troca de mensagens, no entanto oferece uma série de vantagens de um ponto de vista de custo, escalabilidade e controle de condições de disputa.

A comunicação por troca de mensagens é explicita, uma vez que uma mensagem é enviada de um nodo para outro, que precisa estar preparado para recebê-la.

3.1.5 Localização da Memória

Para ambas as classes de comunicação é possível utilizar memória centralizada ou memória distribuída. Esta característica diz respeito à localização da memória em relação aos processadores. Memória centralizada significa que toda a memória do sistema está situada em um mesmo local, sendo acessada por intermédio do meio de conexão que pode ser tanto um barramento quanto uma rede. Neste modelo, o tempo de acesso à memória é igual em todos os processadores. Memória distribuída significa que partes diferentes da memória estão mais próximas de um ou outro processador causando tempos de acesso não uniformes já que acessos à memória remota requer um tempo maior do que o acesso à memória local. As figuras 3.3 e 3.4 mostram exemplos de diagramas de maquinas com memória centralizada e distribuída, respectivamente.

Figura 3.3: Multiprocessador com memória centralizada.

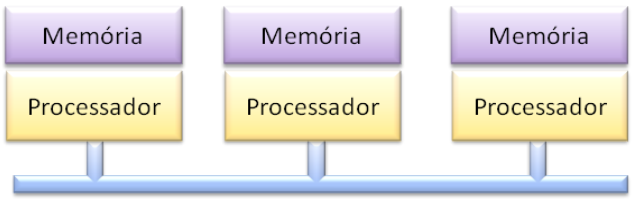


Figura 3.4: Multiprocessador com memória distribuída.

3.1.6 Estado da Arte e Tendências Futuras

Atualmente, praticamente todos os processadores focados ao mercado de computadores pessoais utilizam uma plataforma de múltiplos núcleos, sendo possível encontrar com bastante facilidade processadores de até 6 núcleos.

A arquitetura mais amplamente utilizada é a UMA, por apresentar desempenho satisfatório para esta quantidade de núcleos. Há ainda, uma forte tendência ao uso da tecnologia *Simultaneous Multithreading* (SMT) (DORAI, YEUNG e CHOI, 2003) em cada um dos núcleos, dobrando o número de processadores lógicos disponíveis para a paralelização de tarefas.

O motivo para esta evolução para multiprocessadores é o fato da indústria ter se aproximado rapidamente dos limites físicos no aumento da velocidade do relógio dos processadores, de forma que o aumento do número de núcleos nos

processadores é a alternativa vista como viável para que se mantenha a progressão no aumento do poder de processamento das máquinas pessoais.

De acordo com (FOSTER, 1995) é bastante provável que o número de processadores continuará a aumentar, dobrando a cada ano ou dois, desta forma, softwares devem ser produzidos com a expectativa de que haja um aumento substancial na quantidade de processadores disponíveis para sua execução durante seu período de vida. Neste contexto, o desenvolvimento de software escalável que tire proveito desta evolução, é essencial para que o investimento seja protegido. Um programa feito para executar em apenas um processador ou em um número fixo de processadores é, portanto, um programa ineficiente.

A hipótese defendida no parágrafo anterior é reforçada pelos programas de pesquisa das fabricantes de microchips que estão focando esforços exatamente nesta direção. A Intel, hoje a maior fabricante de processadores para computadores pessoais, possui um programa chamado *Tera-scale Computing Research Program* (INTEL, 2010) cuja visão é avançar na tecnologia de processadores com múltiplos núcleos de forma a obter processadores com centenas de núcleos durante esta década, aumentando o poder de processamento para trilhões de operações por segundo (Teraflops) sobre trilhões de bytes (Terabytes).

3.2 Software Paralelo

Assim como em hardware, o paralelismo em software pode ser classificado por diversos critérios, como: o nível de paralelismo explorado, o modelo de comunicação utilizado e a maneira como ele é executado.

Embora não exista uma classificação aceita comumente pela comunidade de processamento paralelo, vários conceitos e jargões são amplamente utilizados e por isto serão apresentados nesta seção. Além disto, existe um tipo bastante especial de software, o sistema operacional, que visa prover infra-estrutura para o desenvolvimento de aplicações destinadas ao usuário final de um sistema. Dentro desta infra-estrutura estão alguns componentes chave para a execução de software paralelo, sendo importante tomar conhecimento destes.

Quando se considera o contexto de software paralelo, a taxonomia de Flynn pode ser estendida, adicionando quatro novas arquiteturas, conhecidas como *Multiple Program stream/Multiple Data stream* (MPMD), *Single Program stream/Multiple Data stream* (SPMD), *Single Program stream/Single Data stream* (SPSD) e *Multiple Program stream Single Data stream* (MPSD). Esta ultima não tem validade prática pelo mesmo motivo de MISD. A diferença desta classificação para a original é a divisão do fluxo de controle em dois níveis, instrução e programa (MATTSON, SANDERS e MASSINGILL, 2004).

SPMD, portanto consiste em um mesmo programa operando sobre vários conjuntos de dados. Como o nível de controle abordado é o de programa, é possível que uma aplicação do tipo SPMD execute sobre uma arquitetura de hardware do tipo SISD, SIMD ou MIMD. Ou seja, um mesmo programa que opera sobre conjuntos de dados distintos, de forma paralela, pode possuir (e em geral possui) diversos fluxos de instruções distintos. Esta classe contém a maioria absoluta das aplicações paralelas da atualidade, como, por exemplo, os programas que fazem

uso de múltiplas *threads* e que serão estudados com mais detalhe nos próximos capítulos.

MPMD são múltiplos programas operando sobre dados distintos. Exemplos comuns que se enquadram nesta categoria são as aplicações do tipo cliente servidor como os sistemas web, onde vários clientes executam programas distintos em suas máquinas sobre conjuntos de dados distintos. Outro exemplo são os sistemas multi-agentes heterogêneos, onde cada agente é um programa distinto executando sobre dados próprios.

SPSD é uma classe que existe apenas para fins de completude, pois, por considerar apenas um programa em um conjunto de dados, o paralelismo explorado nesta classe só pode ser obtido em nível de instrução.

Alguns termos bastante utilizados na área de processamento paralelo serão apresentados a seguir para facilitar a compreensão do restante do trabalho. O significado de cada termo é definido com base na terminologia utilizada em (MATTSON, SANDERS e MASSINGILL, 2004).

Tarefas são conjuntos, ou blocos de instruções que operam de maneira seqüencial dentro de um programa paralelo.

Unidades de Execução (UE) são, dentro do contexto deste trabalho, processos, ou *threads*.

Processos são recursos como memória, registradores, conjuntos de permissões e E/S providos pelo sistema operacional para execução de uma tarefa. ***Threads*** podem ser vistas como extensões do conceito de processos. Ambos estes termos serão descritos com mais detalhes na próxima seção.

Elemento de Processamento (*Processing Element* ou PE) é uma peça de hardware capaz de executar instruções de uma UE, podendo ser um núcleo de um SMP, ou um computador inteiro em um cluster.

3.2.1 Sistemas Operacionais para Multiprocessadores

A enorme diferença entre a velocidade de processamento e a velocidade de acesso a dados via dispositivos de Entrada e Saída (E/S) em computadores pessoais gerou, muito antes da existência de multiprocessadores, a necessidade para a execução de mais de um programa ao mesmo tempo, de maneira que enquanto um programa aguarda pela resposta de um dispositivo de E/S, outro programa é executado no processador que de outra maneira ficaria ocioso. Para este fim foi criado o conceito de Processos, que são em sua essência programas em execução ao mesmo tempo em um Sistema Operacional.

Pelo mesmo motivo citado anteriormente, passou a ser interessante que um mesmo processo pudesse ser dividido de maneira a executar simultaneamente em um computador. Com o advento dos multiprocessadores esta necessidade foi reforçada, para permitir a execução de um mesmo programa em mais de um processador simultaneamente. Para isto foi criado o conceito de Thread.

O motivo para descrever sistemas operacionais, neste trabalho, vem das diferentes expectativas que um programa e um sistema operacional possuem quanto ao uso de processamento paralelo. Em um sistema operacional, a existência de concorrência é inerente da sua condição, já que é esperado que várias tarefas

independentes sejam realizadas ao longo do tempo, por diferentes usuários, logo, o seu objetivo é aumentar o fluxo de saída e o tempo de resposta destas tarefas além de garantir o compartilhamento de recursos de forma segura e sem interferência. Ao contrário, em programas paralelos, explorar concorrência é um desafio, e o seu objetivo é a diminuição do seu tempo de execução. A interferência entre recursos compartilhados é muitas vezes desejável e necessária.

3.2.2 Processos

Processos são fluxos seqüenciais em execução juntamente com seu contador de execução, seus registradores e variáveis, ou seja, processos são programas em execução.

Hoje em dia praticamente todos os sistemas operacionais modernos possuem suporte à execução de múltiplos processos ao mesmo tempo, de forma que enquanto um processo executa uma operação lenta (acesso a disco, por exemplo), outro processo pode ocupar o processador que de outra maneira ficaria ocioso.

3.2.3 Threads

Threads são ramificações de um processo em mais de um fluxo de controle. Ou seja, um mesmo programa com seu espaço de endereçamento único pode possuir mais de um fluxo de execução de código, realizando diferentes tarefas ao mesmo tempo. Cada um destes fluxos de execução é chamado de *thread*. Com o surgimento dos multiprocessadores, uma importante evolução é o suporte a execução de *threads*, onde um mesmo processo pode possuir fluxos paralelos de execução e ocupar mais de um processador ao mesmo tempo.

De forma similar ao processo, *thread* é um conceito bastante difundido e suportado em praticamente todos os sistemas operacionais modernos. A figura 3.5 ilustra dois programas equivalentes. O programa A não utiliza *threads*, possuindo apenas um fluxo de execução. O programa B utiliza duas *threads*, dividindo as operações em dois fluxos de execução e consumindo apenas a metade do tempo de A para executar a mesma quantidade de operações.

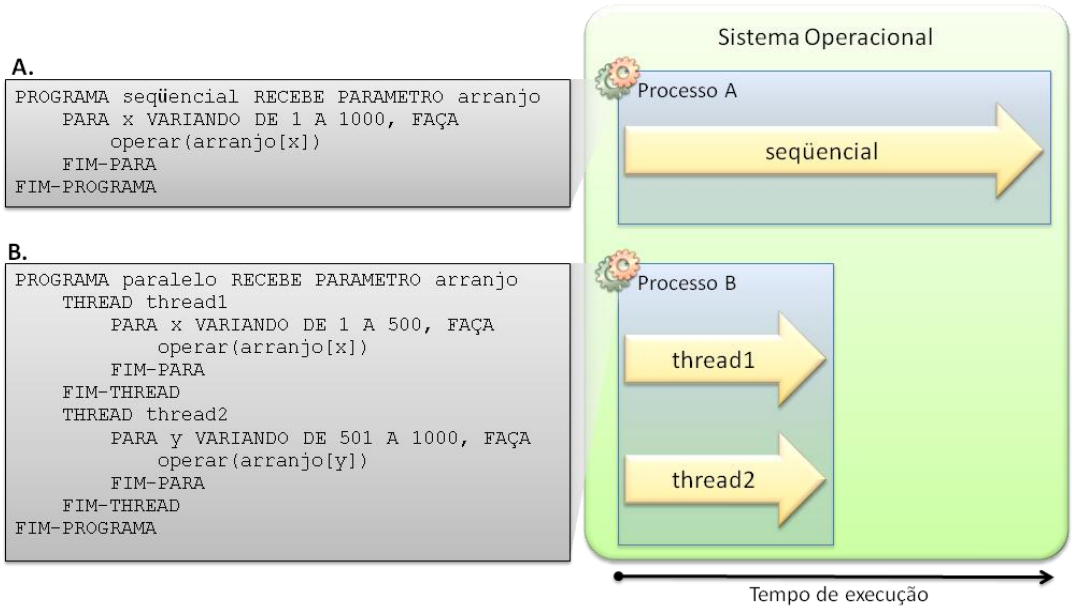


Figura 3.5: Dois programas equivalentes, com e sem *threads*.

4 PROGRAMAÇÃO PARALELA

Neste capítulo será apresentada uma visão geral da programação paralela, os desafios enfrentados para sua utilização e os mecanismos utilizados para solucioná-los.

Programação paralela e projeto de software para sistemas paralelos tem sido uma área bem estabelecida na ciência da computação há tanto tempo quanto o projeto e desenvolvimento de hardwares paralelos, uma vez que é praticamente inviável obter através do hardware sozinho, a execução paralela de um programa.

Quando comparada com a programação seqüencial, é possível perceber que programas paralelos são muito mais complicados. Em programação seqüencial, geralmente, simples é sinônimo de eficiente, de forma que a expectativa é que um bom programador produza o código mais simples possível. Programar para processamento paralelo de forma quase oposta, não é programar para eficiência, nem programar para portabilidade, nem para escalabilidade e nem mesmo para simplicidade. Neste universo, cada um destes aspectos conflita com os demais fazendo com que a programação paralela precise encontrar um balanço entre estas características que seja capaz de satisfazer a um determinado problema.

Por fim, programar para multiprocessadores envolve, além dos desafios da programação seqüencial, uma série de novos desafios que tornam necessário o uso de ferramentas eficazes de modelagem e programação para que seja possível satisfazer as expectativas de um programa paralelo, em especial, que seu desempenho seja sempre superior ao da sua versão seqüencial.

A seguir os principais desafios são apresentados, seguidos das ferramentas mais utilizadas para contorná-los ou resolvê-los.

4.1 As Dificuldades no uso de Processamento Paralelo

Suponha-se um computador com número infinito de processadores, onde cada processador acessa a memória de forma independente e sem atrasos. Agora, imagine-se que o programa executando neste computador é composto por um conjunto de instruções completamente independentes que podem ser executadas numa ordem qualquer, operando sobre uma quantidade arbitrária de dados não relacionados.

Esta situação hipotética ilustra um cenário ideal ao paralelismo. No mundo real, há restrições em todos estes requisitos em praticamente todas as aplicações que se analise, seja de um ponto de vista de hardware, onde o número unidades de

processamento, o número de canais de acesso aos dados e a velocidade destes canais são todos finitos, ou de um ponto de vista de software, onde operações dependem umas das outras seja pela seqüência do seu fluxo de controle ou pela seqüência do seu fluxo de dados.

Para classificar os problemas relacionados ao paralelismo, é importante considerar alguns pontos. O primeiro é que existem tipos diferentes de paralelismo.

De acordo com (AKENINE-MÖLLER, 2008) um sistema paralelo pode ser dividido em dois métodos: paralelismo temporal e paralelismo espacial.

O paralelismo espacial ou paralelismo real consiste em dividir o problema de forma que cada processo produza resultados de maneira independente dos demais. Este tipo de paralelismo ocorre geralmente em aplicações independentes e não é simples obtê-lo de forma balanceada, onde as partes paralelizáveis possuem um tempo de execução similar. O paralelismo espacial se divide ainda em paralelismo de dados e paralelismo de tarefas ou funcional.

O paralelismo temporal consiste em modelar o problema de maneira semelhante a uma linha de produção, onde cada processo resolve uma parte do problema e passa os resultados para o processo seguinte.

Outro ponto a considerar é que o paralelismo, de qualquer tipo, se manifesta em diversos níveis. Ele vai desde a quantidade de bits operada simultaneamente por uma instrução até a execução de múltiplos processos ao mesmo tempo em computadores distintos ao redor do mundo.

Uma maneira adequada para classificar estes níveis é considerar a granularidade das unidades de trabalho enviadas aos processadores que são candidatas à execução em paralelo. A tabela 4.1 mostra algumas granularidades juntamente com o elemento ou individuo responsável por tornar a sua execução paralela.

Tabela 4.1: Classificação dos níveis de paralelismo em granularidades diferentes (MOURA E SILVA e BUYYA, 1999).

Granularidade	Unidade de trabalho	Paralelizada por
Muito fina	Instrução	Processador
Fina	Loop/Bloco de instruções	Compilador
Média	Função/Método	Programador
Grossa	Processo/Thread	Programador

O nível tratado neste trabalho é aquele onde o programador pode atuar, portanto, embora os problemas apresentados permeiem as diferentes escalas, as soluções que serão apresentadas serão orientadas principalmente à granularidade média e grossa.

O desenvolvimento de sistemas paralelos adiciona problemas que não estão presentes em sistemas seqüenciais, os quais são (MOURA E SILVA e BUYYA, 1999):

- Não-determinismo;

- Sincronização;
- Divisão e distribuição de dados;
- Balanço de carga;
- Tolerância a falhas;
- Heterogeneidade;
- Memória compartilhada ou distribuída;
- Deadlocks e condições de disputa;
- Comunicação;

Cada um destes problemas adiciona desafios próprios, de forma que hoje em dia apenas poucos programadores possuem conhecimento para utilizar sistemas paralelos para executar código com qualidade de produção.

A seguir cada um destes problemas será observado em mais detalhes, juntamente com as técnicas utilizadas para resolvê-los ou contorná-los.

4.1.1 Não-determinismo

Determinismo significa que uma determinada ação terá sempre a mesma conseqüência, não interessando a situação em que esta ação é realizada. Dentro da computação, um sistema determinístico é aquele onde uma mesma entrada resultará sempre em uma mesma saída. Não-determinismo é a ausência desta característica, ou seja, para um mesmo conjunto de dados de entrada, não é possível determinar com certeza qual será a sua saída.

Em alguns casos, o não-determinismo se manifesta em situações que não influenciam a utilidade dos resultados de um sistema, sendo permitido e eventualmente desejado. Como exemplo, suponha um carrinho de compras tendo seu conteúdo processado por um caixa de supermercado. Neste caso, não importa a ordem em que as compras são passadas pelo caixa, contanto que o sejam, portanto o não-determinismo pode ser ignorado.

Quando não desejado, no entanto, o não-determinismo é um problema, tornando impossível garantir os resultados de um programa. Para benefício dos programadores, existem técnicas que permitem garantir o determinismo. Este pode ser garantido de forma estática, ou seja, forçando que a escrita de código seja correta, ou dinamicamente, em tempo de execução. A forma mais utilizada é a estática, como, por exemplo, fork/join e Futuros (WELC, JAGANNATHAN e HOSKING, 2005) que procuram garantir uma semântica seqüencial utilizando um modelo de desempenho paralelo.

4.1.2 Sincronização

A sincronização do fluxo de controle de tarefas executadas em paralelo é uma necessidade derivada de outros problemas de paralelismo. Sempre que for preciso que uma tarefa aguarde por informações de outra, será necessário um mecanismo para que elas se comuniquem, e possam, então, alterar seus estados em função dos seus resultados, ou seja, entrem em sincronia. Os tipos comuns de problemas decorrentes da falta de sincronismo são as situações onde a execução intercalada de operações gera conflitos de dados das tarefas. Estes conflitos são conhecidos pelos

nomes: Leitura após Escrita (RAW), Escrita após Escrita (WAW) e Escrita após Leitura (WAR).

Primitivas fork/join, são exemplo de técnica que pode ser utilizada para garantir explicitamente a sincronização entre tarefas. Esta técnica, bem como grande parte das técnicas existentes para resolver este problema, é uma ferramenta de baixo nível, cabendo ao programador modelar o programa de forma adequada para utilizá-la de forma correta.

4.1.3 Divisão e Distribuição de Dados

Dividir dados para que possam ser tratados de forma paralela caracteriza aquilo que é comumente chamado de paralelismo de dados (HILLIS e STEELE, 1986). As técnicas existentes permitem obtê-lo de forma automática, transparente ao programador ou de maneira explicita, ou manual. A divisão e distribuição implícita é feita por meio de linguagens especiais de programação, conhecidas como data-parallel languages como NESL, HPF (HAYASHI e SUEHIRO, 2008) e Manticore (FLUET, RAINEY, *et al.*, 2007) geralmente utilizadas para fins bastante específicos, como a melhoria de desempenho de aplicações legadas.

Realizar a divisão e distribuição dos dados de forma manual, requer, em essência, uma boa compreensão do conceito envolvido neste tipo de paralelismo e o uso de bibliotecas destinadas a este tipo de paralelismo, como Threading Building Blocks da Intel ou OpenMP.

4.1.4 Balanço de Carga

O balanço de carga é um dos principais problemas para a aplicação de paralelismo de tarefas em dispersão temporal (pipeline), já que neste tipo de paralelismo, uma etapa mais lenta do que as demais fará com que as outras etapas a fiquem aguardando. Isto irá gerar sobrecarga das etapas anteriores, enfileirando suas tarefas, e ociosidade das posteriores, deixando-as sem dados para trabalhar até que os recebam da tarefa mais lenta. Ou seja, para este tipo de paralelismo é essencial que uma grande tarefa seja dividida em partes balanceadas, de forma que cada parte ocupe um período similar de tempo para ser executada. A figura 4.1 apresenta o resultado de um programa paralelo onde todas as tarefas precisam terminar para que o programa possa prosseguir. O tempo final do programa será o tempo da tarefa mais lenta já que as tarefas mais rápidas ficarão aguardando pelo seu fim.

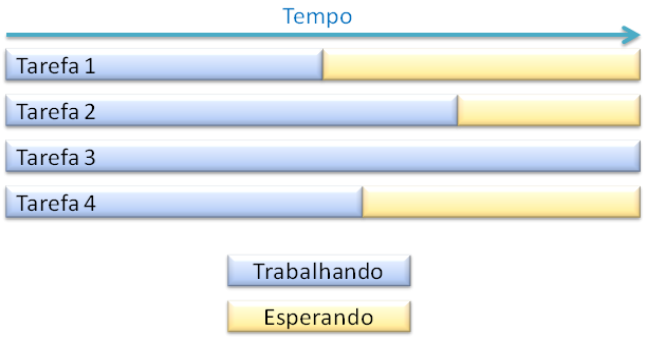


Figura 4.1: Tempo de execução e espera de quatro tarefas desbalanceadas executando em paralelo. O tempo total de execução é o tempo da tarefa mais lenta.

Algumas bibliotecas como Cilk++ (LEISERSON e MYRMAN, 2008) trabalham com balanceamento dinâmico de carga por meio de um agendador de tarefas capaz de avaliar e distribuir tarefas de acordo com a disponibilidade de hardware. Isto, evita o desperdício de recursos que ficariam ociosos em função de tarefas em espera alocando o processamento disponível a outras tarefas independentes. No entanto, o agendador de tarefas não é capaz de endereçar o problema de fluxo de saída em programas estruturados em dispersão temporal. Balancear corretamente a carga destinada a cada tarefa, na maior parte dos casos, é uma atividade que cabe inteiramente ao programador, distribuindo de forma adequada o trabalho entre elas sempre que possível.

4.1.5 Condições de Disputa

As condições de disputa são causadas por acesso concorrente a recursos compartilhados podendo ocasionar uma interferência não desejada entre dois ou mais processos. Como os recursos de um computador são limitados, situações em que tarefas competem por eles são bastante comuns. Situações de disputa que não forem tratadas adequadamente podem ocasionar corrupção de dados e comportamento não-determinístico indesejado.

Há duas maneiras para tratar este problema, a primeira é replicando recursos compartilhados sempre que possível. De um ponto de vista de software, isto é possível utilizando um mecanismo de replicação de dados compartilhados e de troca de mensagens para mantê-los coerentes. Em boa parte das situações reais, no entanto é necessário utilizar uma segunda maneira, que consiste em sincronizar o acesso aos recursos compartilhados fazendo uso de técnicas de exclusão mútua. Estas técnicas evitam anomalias decorrentes da execução intercalada de partes não atômicas de dois ou mais processos que compartilhem um mesmo recurso, como por exemplo, os conflitos de dados RAW, WAW e WAR.

Técnicas de seção crítica, como semáforos e monitores e variáveis Mutex, presentes em praticamente todas as linguagens de programação modernas, fazem com que este problema possa ser contornado de forma bastante simples, porém ainda cabe ao programador definir seções críticas sem que o desempenho seja comprometido.

Imagine-se, por exemplo, um programa composto por diversas *threads*, todas realizando operações sobre uma mesma variável. Supondo que o código para realizar estas operações se situa em uma seção crítica, o resultado, em termos de desempenho deste sistema será ruim, já que as operações não irão ocorrer em paralelo em hipótese alguma.

O caso citado como exemplo é extremo, servindo apenas para ilustrar o que pode ocorrer se for feito mal uso de seções críticas, demonstrando que estas devem ser evitadas ao máximo, e que à medida do possível deve-se procurar fazer com que poucas *threads* estejam em uma seção crítica simultaneamente, evitando comportamento serial sempre que possível.

4.1.6 Deadlocks

Deadlocks também estão associados a acesso a recursos compartilhados e exclusão mútua, mais especificamente à situação onde dois ou mais tarefas

precisam de mais de um recurso não preemptível ao mesmo tempo e cada uma delas conseguiu se apoderar de um destes recursos. Assim, um processo ficará aguardando o outro liberar o seu recurso indefinidamente. Existem técnicas para detectar e recuperar deadlocks, mas nenhuma das técnicas é completamente eficaz. Elas adicionam custo ao processamento, e, em boa parte das vezes, os deadlocks são simplesmente ignorados (TANENBAUM, 2003). Devido a estas limitações, ainda cabe ao programador, desenvolver uma aplicação livre de *deadlocks*.

4.1.7 Heterogeneidade

Heterogeneidade consiste em se ter nodos com diferentes características (velocidade de processamento, tamanho de memória, largura de banda do canal de comunicação) executando uma aplicação paralela. Não é uma tarefa simples desenvolver um programa que se comporte bem em sistemas heterogêneos, já que o balanceamento de carga entre os nodos precisa ser ajustado em função das características de cada nodo para evitar que um nodo fique aguardando pelos resultados de outro mais lento.

Para grande parte dos multiprocessadores da atualidade, a heterogeneidade não constitui um problema imediato já que as máquinas com múltiplos núcleos são bastante homogêneas por compartilharem praticamente todas as suas características, porém a tendência futura é de que este problema aumente de proporção e não possa ser ignorado. Em especial, isto deve ocorrer à medida que o número de núcleos dos processadores cresce e que passam a existir máquinas com uma diferença considerável em suas características físicas.

No universo dos sistemas distribuídos, este é um problema constante e precisa ser tratado sempre, sob o risco de causar sérios impactos no desempenho.

Há técnicas, como a proposta em (GODFREY e KARP, 2006), onde a heterogeneidade de cada nodo em um sistema computacional distribuído é precificada. Esta técnica atribui um valor a cada um destes nodos que representa o desempenho deste nodo. Este valor permite à aplicação ajustar a distribuição de carga entre os nodos dinamicamente de acordo com o seu desempenho.

4.1.8 Tolerância a Falhas

Processamento paralelo é geralmente enunciado como sendo uma solução de um ponto de vista de tolerância a falhas, porém existem alguns problemas deste campo que são criados ao se trabalhar com uma aplicação paralelizada. O principal é garantir que falhas no canal de comunicação entre os processos não invalide o sistema como um todo, ou seja, fazer com que o mecanismo de comunicação seja tolerante a falhas. Este é um problema que afeta, de forma mais ampla, sistemas distribuídos que se intercomunicam por troca de mensagens, e a tolerância a falhas é geralmente deixada ao encargo do protocolo e da infra-estrutura de comunicação, onde são aplicadas técnicas de comunicação com confirmação, re-envio de dados e uso de canais redundantes entre outras.

4.1.9 Comunicação

Este é, sem sombra de dúvida, o maior dos problemas, sendo geralmente o ponto onde os gargalos de desempenho se situam.

Para que seja possível trocar informações entre duas tarefas que executam em paralelo, é necessário o estabelecimento de um canal por onde estas informações

possam trafegar. Cada aplicação possui requisitos próprios e muitas vezes singulares com relação à comunicação. Por exemplo, enquanto para uma aplicação é aceitável receber dados fora de ordem, para outra pode ser um requisito que os dados venham em seqüência. Em outro cenário, enquanto uma aplicação pode tolerar a perda de alguns dados durante a comunicação, outra requer que todos os bits sejam corretamente entregues.

Para cada necessidade, existem um ou mais mecanismos que podem ser empregados, porém existem algumas técnicas que se sobressaem e estão presentes em praticamente todas as aplicações paralelas. As técnicas mais utilizadas no que diz respeito à comunicação entre threads são: troca de mensagens, que se baseia nas primitivas *send/receive*, e, memória compartilhada, que se baseia nas primitivas *load/store*.

O mecanismo de troca de mensagens exige a definição prévia da estrutura de comunicação por parte do programador, ou seja, torna a comunicação explícita. O modelo de memória compartilhada deixa a comunicação implícita, já que não exige que uma *thread* conheça as demais threads que dependem das informações providas por ela.

Para cada um destes mecanismos, existem técnicas orientadas a computadores distribuídos (BUSI, GORRIERI e ZAVATTARO, 2000) ou não distribuídos (BUNTINAS, MERCIER e GROPP, 2007), o que sem dúvida facilita ao desenvolvedor o estabelecimento de um canal adequado para a troca de informações. Deve-se, no entanto, salientar que independente da técnica utilizada, a comunicação só deve ser utilizada quando não for possível evitá-la, uma vez que o uso do canal de comunicação sempre causará atrasos, seja pela sincronização, no caso da memória compartilhada, ou pela velocidade do canal para a troca de mensagens. Por este motivo, cada tarefa precisa ser modelada de maneira a focar seu trabalho nos dados que estão presentes em sua memória local. A esta característica se da o nome de localidade (ATTIYA, 2007), e a sua garantia ainda é tarefa do programador.

4.2 Modelos de Programação Paralela

Para ajudar a resolver os problemas citados anteriormente, existem modelos e padrões de programação que procuram prover, respectivamente, um ambiente para a modelagem e desenvolvimento de software e um conjunto de práticas eficazes e consolidadas.

Esta seção apresenta uma visão geral dos modelos utilizados atualmente para a programação paralela. Seu conteúdo é baseado, principalmente, nos trabalhos desenvolvidos em (RAUBER e RÜNGER, 2010) e (MOURA E SILVA e BUYYA, 1999).

A necessidade da definição de um modelo de programação vem da enorme influência que o sistema computacional exerce sobre esta atividade. O sistema computacional consiste no conjunto de componentes de software e hardware que são providos ao programador. Os aspectos de hardware são definidos pela arquitetura do computador paralelo, os aspectos de software são definidos pelo

ambiente de software incluindo o sistema operacional, a linguagem de programação e as bibliotecas de programação (MATTSON, SANDERS e MASSINGILL, 2004).

Um **modelo de programação** é a abstração sobre o sistema computacional que representa a visão do programador a respeito da máquina, permitindo ao programador expressar um algoritmo paralelo.

Existe uma série de características que podem ser utilizadas para diferenciar os modelos de programação. Entre estas características estão: o nível de paralelismo explorado, a especificação implícita ou explicita do paralelismo, a maneira como as partes paralelizáveis são definidas, o modo de execução das unidades paralelas, o padrão de comunicação utilizado e os mecanismos de sincronização utilizados

Dentre todas estas características, aquela que diferencia de forma mais saliente um modelo de programação é o padrão de comunicação utilizado, de forma que os modelos mais comuns são abstrações baseadas nos mecanismos de comunicação das arquiteturas paralelas de hardware tradicionais, sendo eles: memória compartilhada e troca de mensagens.

Em computadores seqüenciais, o modelo de programação utilizado é o de Von Neumann. Como todos os computadores seqüenciais são baseados neste modelo, é razoável considerar que um programa que utiliza esta abstração, será facilmente mapeado para qualquer máquina seqüencial.

Em sistemas paralelos, no entanto, a diversidade de combinações entre ambientes de hardware e software afeta diretamente a visão do programador a respeito do sistema, tornando possível que uma variedade de diferentes programas seja desenvolvida para a implementação de um mesmo algoritmo. Embora a variedade não seja, necessariamente, um ponto negativo, modelos de programação atrelados a uma arquitetura de hardware específica tornam o software produzido pouco portável a diferentes plataformas, além de restringir a vida útil possível do software, que seria, geralmente, muito maior do que a do hardware.

Estes pontos fizeram que, durante a década de 90, os programadores percebessem a necessidade de ambientes de software que abstraiam a arquitetura de hardware. Em função disto surgiram dezenas de ambientes de software voltados ao paralelismo. Na atualidade, a comunidade de programação paralela convergiu para o uso de apenas dois ambientes, OpenMP que utiliza o paradigma de memória compartilhada e MPI baseado em troca de mensagens.

4.2.1 Modelos de Programação baseados em memória compartilhada

Neste tipo de modelo de programação, as UEs compartilham áreas de memória, sendo possível a cada tarefa modificar os dados utilizados por outras diretamente, sem necessidade de conhecer a origem ou o destino dos dados processados. Uma vez que não é preciso que uma tarefa tenha conhecimento sobre outras eventualmente acessando os mesmos dados, este tipo de comunicação é considerado implícito.

Em muitos casos, a arquitetura física do computador utiliza memória privada, porém, mesmo neste tipo de máquina é possível desenvolver programas modelados com o paradigma de memória compartilhada. Pelo fato de a memória compartilhada ser uma abstração de software, o modelo de programação neste caso

pode ser chamado memória compartilhada virtual (MOURA E SILVA e BUYYA, 1999).

4.2.2 Modelos de Programação baseados em troca de mensagens

Alternativamente ao uso de memória compartilhada, a comunicação entre tarefas em um sistema que faz uso de processamento paralelo pode ser feita através da passagem de mensagens (DONGARRA, OTTO e SNIR, 1996). Esta abordagem facilita o desenvolvimento dos sistemas e sua escalabilidade para um número maior de processadores.

O paradigma de troca de mensagens em software trabalha com o mesmo conceito que seu homônimo em hardware e requer comunicação explicita, cabendo ao programador o estabelecimento prévio da topologia da rede.

4.2.3 Estado da Arte em Modelos de Programação Paralela

Hoje não há um modelo de programação genérico que enderece todos os problemas de programação de forma satisfatória. Embora alguns modelos considerem subconjuntos destas características em maior ou menor grau ainda é responsabilidade do programador satisfazê-las.

Grande parte das técnicas empregadas para a resolução de problemas relacionados ao paralelismo está disponível em linguagens de programação voltadas ao paralelismo de threads ou em bibliotecas de baixo nível que definem o ambiente de software do sistema computacional.

Dentre as linguagens, duas apresentam a mais vasta aplicação na indústria de software, são elas Java e C#. Java possui mecanismos nativos para suporte a threads e sincronização sem a necessidade do uso de bibliotecas adicionais (OAKS e WONG, 2004). Nesta linguagem há um conjunto de classes (pacote) chamado *java.util.concurrent* que provê estruturas de apoio para comunicação e coordenação oferecendo suporte a modelos de programação baseados em memória compartilhada e memória virtual. Por outro lado, Java é uma linguagem famosa por sua ampla portabilidade, obtida em detrimento de seu desempenho. Porém essa linguagem tem evoluído rapidamente em termos de desempenho, e é esperado que esta característica negativa deixe de ser um problema ao longo do tempo.

Quanto a bibliotecas de programação, as duas siglas, OpenMP e MPI referenciam as especificações para bibliotecas de processamento paralelo mais difundidas na comunidade de programação paralela. Diferentemente da linguagem Java onde as unidades de execução são de conhecimento do programador, OpenMP e MPI proporcionam modelos de programação que abstraem a UE, deixando ao encargo do programador apenas a definição das tarefas paralelizáveis.

Algumas outras bibliotecas menos utilizadas possuem bastante importância, seja na comunidade cientifica, seja na indústria de software, por isto serão mencionadas.

O modelo de coordenação LINDA (DI GIUSTO e GABBRIELLI, 2008) consiste em uma plataforma de abstração da arquitetura do computador paralelo. Seu foco é prover um modelo de programação baseado em memória compartilhada que pode ser utilizado em máquinas com espaços de endereçamento privado.

Existem, também, bibliotecas dedicadas à conversão de algoritmos seqüenciais em algoritmos paralelos com a intenção de melhorar o desempenho destes. Por fim, boa parte das bibliotecas que atuam em um nível mais alto de modelagem do sistema é voltada ao paralelismo de dados, por este ser o tipo de paralelismo mais adequado para o uso em clusters. O foco neste tipo de paralelismo pode ser percebido nas bibliotecas *Threading Building Blocks* da Intel (REINDERS, 2007) e TPL da Microsoft entre outras.

O uso destes recursos facilita o desenvolvimento, porém, não há garantia de que a solução produzida tire o proveito esperado do paralelismo. Além disto, o uso excessivo de comunicação, sincronização e sessões críticas pode comprometer seriamente o desempenho de um sistema, e a adaptação de algoritmos seqüenciais gera muitas vezes soluções hibridas que alternam entre seqüencial e paralelo, deixando de explorar todo o potencial das máquinas paralelas.

Sendo assim, mesmo com um modelo de programação adequado, tornar a resolução de um problema eficaz, requer que a mentalidade do programador esteja voltada à programação paralela, e é neste contexto que os padrões de projeto para programação paralela são úteis. Eles permitem que um programa seja elaborado desde a sua concepção levando em conta os problemas e as necessidades da execução paralela.

4.3 Padrões de Projeto para Programação Paralela

Padrões de projeto são boas soluções aplicadas em problemas recorrentes em um contexto específico (GAMMA, 2003). Dentro da programação paralela, vários padrões de projeto são encontrados, e uma vez bem definidos, tornam-se uma ferramenta poderosa para o desenvolvimento de sistemas capazes de atender as expectativas de um programa paralelo. Entre estas expectativas se destacam:

- Tornar o programa independente da arquitetura de hardware e portável a diferentes arquiteturas;
- Tornar a comunicação entre tarefas transparente para o desenvolvedor;
- Produzir um programa simples e robusto;
- Tornar o programa tolerante a falhas;
- Produzir um programa de fácil manutenção, baseado em linguagens e bibliotecas tradicionais;
- Produzir um programa com alto desempenho;
- Tornar o paralelismo transparente para o programador;

Um dos melhores materiais disponíveis para o estudo deste tópico se encontra no livro (MATTSON, SANDERS e MASSINGILL, 2004) onde os padrões de projeto são definidos e organizados por meio de uma linguagem de padrões que explica o fluxo de aplicação juntamente com as decisões envolvidas na escolha de cada um dos padrões pelo programador. Nas seções subseqüentes, esta linguagem e os padrões mais relevantes a este trabalho, serão descritos em mais detalhe.

4.3.1 Linguagem de Padrões

Uma linguagem de padrões é uma ferramenta que classifica os padrões de projeto em uma estrutura hierárquica que representa um fluxo de decisões destinado

a guiar o seu usuário. Desta forma, ela permite que um programador modele sistemas complexos a partir do uso desses. Uma das principais características da linguagem de padrões para processamento paralelo é a divisão dos padrões em quatro espaços distintos de projeto, cada espaço correspondendo a uma das etapas da definição de um programa paralelo.

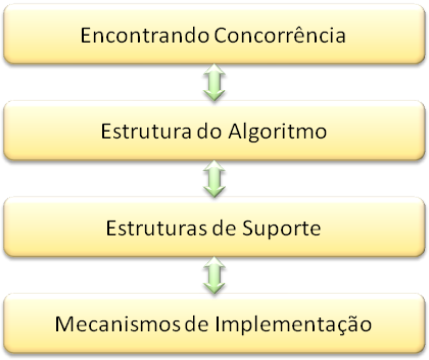


Figura 4.2: Os quatro espaços de projeto da linguagem de padrões.

O fluxo descrito na figura 4.2 sugere que o programador realize a modelagem do sistema seguindo passos que permitem explorar o paralelismo de maneira adequada. Os passos consistem em definir a melhor maneira para encontrar porções paralelizáveis, estruturar o sistema para que as partes paralelizáveis possam ser executadas de maneira concorrente, escolher as estruturas de suporte à execução paralela que mais condizem com o algoritmo definido e por fim, escolher quais os mecanismos de implementação que permitem transformar estas estruturas de suporte em programas executáveis.

4.3.2 Encontrando concorrência

Para este espaço de projeto, a linguagem de padrões define a utilização dos padrões de projeto da seguinte forma. Primeiro, o problema precisa ser decomposto em tarefas paralelas e em dados paralelos. Em problemas complexos, geralmente há um pouco de cada um destes tipos de paralelismo, porém a manifestação predominante irá determinar qual será a estrutura do algoritmo.

A seguir, as tarefas devem ser agrupadas de acordo com suas dependências, então ordenadas de acordo com as mesmas, e, por fim, deve ser definido o conjunto de dados que precisa ser compartilhado entre as tarefas.

O ultimo passo é a aplicação de um padrão para a revisão do projeto gerado até o momento.

4.3.3 Estrutura do Algoritmo

Para o espaço de projeto anterior, uma seqüência de padrões é aplicada, gerando uma visão das tarefas paralelizáveis dentro do programa em desenvolvimento. No espaço de projeto “Estrutura do Algoritmo”, um padrão para mapear a concorrência em UEs precisa ser escolhido.

Os padrões neste espaço se dividem em três tipos, sendo que, sua utilização depende da maneira pela qual o programador quer organizar a concorrência. A

figura 4.3 demonstra uma visão geral dos padrões existentes neste espaço de projeto.

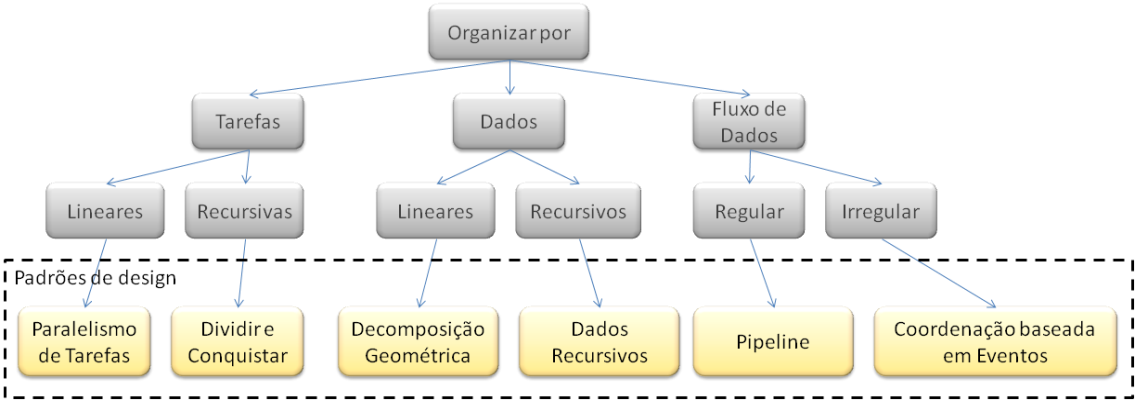


Figura 4.3: Os padrões de projeto do espaço "Estrutura do Algoritmo" e o processo de decisão envolvido na escolha por um deles.

4.3.4 Estruturas de Suporte

Os padrões de estruturas de suporte descrevem construções que suportam a expressão de algoritmos paralelos. Estas construções constituem um passo intermediário entre a estrutura do algoritmo e os mecanismos de implementação do programa.

As estruturas de suporte se dividem em estruturas de programa e estruturas de dados. As estruturas de programa são: SPMD, Mestre/Trabalhador (*Master/Worker*), Paralelismo de Loop, Fork/Join. As de dados são: Dados Compartilhados, Fila Compartilhada, Arranjo Distribuído (*Distributed Array*).

Cada padrão das estruturas de suporte não define uma forma exclusiva para modelagem de um problema, sendo possível, inclusive, implementar um padrão a partir de outro.

4.3.5 Mecanismos de Implementação

Os mecanismos de implementação são padrões que definem a gestão de unidades de execução, a sincronização e a comunicação. Neste nível, os padrões são estabelecidos pelo modelo de programação, de forma que cabe ao programador escolher adequadamente este modelo.

Quanto à gestão de UEs, para ambientes de software baseados em memória privada como MPI, as UEs são mapeadas em processos. Ambientes baseados em memória compartilhada como OpenMP e Java, utilizam threads, escalonadas pelo sistema operacional.

Para sincronização, os padrões utilizados são: Cercas (*Fences*), Barreiras e Exclusão Mútua.

Para comunicação, em sistemas de memória compartilhada, os mecanismos de sincronização são utilizados para garantir acessos limpos à informação compartilhada. Para troca de mensagens, alguns padrões específicos são usados: Ponto-a-ponto, Broadcast/Multicast, Barreiras e Redução. Estes três últimos são direcionados à comunicação coletiva, onde uma única troca de informação envolve mais de duas tarefas.

5 O MODELO DE TAREFAS CONEXIONISTAS (MTC)

Neste capítulo é apresentado um modelo de programação para processamento paralelo inspirado nos conceitos do Processamento Paralelo Distribuído (PDP) apresentado em (MCCLELLAND, RUMELHART e HINTON, 1983). Este modelo provê uma abstração da plataforma de hardware e de software por meio de mecanismos que permitem projetar um sistema paralelo, bem como verificar e refinar o seu desempenho.

O capítulo está organizado de maneira a proporcionar uma visão geral do modelo de programação desenvolvido neste trabalho. A seguir este modelo é detalhado em seus dois principais componentes: um padrão de coordenação e uma biblioteca de mecanismos de implementação desenvolvida em linguagem Java, tendo como objetivo servir como um guia prático para a sua utilização.

A teoria do Processamento Paralelo Distribuído sugere que, o caminho para produção de máquinas inteligentes deve passar pela produção de máquinas e sistemas capazes de processar informação de forma paralela semelhante aos sistemas inteligentes biológicos. Esta conclusão é obtida como resposta para a seguinte questão: *"Porque pessoas são mais espertas do que máquinas?"*.

Ao estudar funcionamento dos mecanismos neurais e cognitivos, o autor propõe um modelo lógico generalizado destes, que pode ser aplicado aos mais diversos problemas da realidade. Este modelo faz parte dos fundamentos do campo da inteligência artificial (IA) conhecido como IA conexionista, onde estão classificadas as redes neurais artificiais. O esquema lógico é uma abstração de um sistema nervoso biológico, e, portanto considera o paralelismo como parte essencial de sua concepção, já que neste há uma infinidade de unidades de processamento e fluxos de dados, todos coordenados para a obtenção de um fim comum – a sobrevivência do ser. Nesta abstração, as unidades de processamento correspondem aos neurônios e para que estes desempenhem seu processamento existem funções, regras e estados auxiliares.

Este esquema considera que as unidades de processamento desempenham tarefas muito simples, e que o conhecimento do sistema é um resultado da rede de interconexões entre estas unidades, e não do processamento realizado pela unidade em si. Neste caso, a atividade de uma unidade de processamento não passa de um simples somatório de valores numéricos, que tomado individualmente não possui valor prático. Embora o modelo lógico não limite as unidades a um nível tão elementar de processamento, este é o nível explorado nas redes neurais artificiais e, portanto, aquele que possui a maior gama de aplicações e estudos na atualidade.

O modelo proposto neste trabalho, experimenta a ampliação da escala das unidades de processamento a um nível onde sua execução individual possua sentido prático, e, desta forma, possa ser modelado de maneira independente das demais.

Para entender a proposta, considere-se o seguinte raciocínio: em um sistema inteligente biológico um único neurônio desempenha uma atividade simples e específica para que um conjunto deles efetue uma atividade mais complexa. Este conjunto de neurônios, por sua vez, desempenha atividades específicas que viabilizam, a um aglomerado de conjuntos de neurônios realizarem outra atividade ainda mais complexa. Supondo que este padrão ocorre repetidas vezes, até que se atinja a escala de um sistema nervoso por inteiro, serão obtidos grupos independentes de neurônios desempenhando atividades com claro valor funcional, como por exemplo, a visão, a fala, a memória, etc. Esta divisão do sistema nervoso biológico em subsistemas funcionais é uma teoria comprovada pela neurociência.

Ou seja, em vez de considerar que uma unidade de processamento é um único neurônio, o modelo proposto considera que uma unidade é um conjunto de neurônios que desempenham uma tarefa específica.

Este modelo utiliza o padrão de projeto chamado **Coordenação Baseada em Eventos,** apresentado no capítulo anterior no espaço de projeto de "Estrutura do Algoritmo". A Coordenação Baseada em Eventos foi escolhida por ser uma analogia quase direta a um modelo conexionista tradicional do Processamento Paralelo Distribuído. Basta, para isso, imaginar que cada tarefa, ao representar um aglomerado de neurônios, é ao mesmo tempo produtora, pois produz estímulos aos aglomerados vizinhos; e também consumidora, pois recebe estímulos daqueles que estão conectados a ela. Os estímulos são, neste caso, os eventos trafegados, e, enquanto não houver um estímulo, uma tarefa simplesmente fica em estado de espera.

Um ponto importante a ser considerado nesta analogia é o fato de um estímulo ser enviado a mais de um receptor ao mesmo tempo, pois um neurônio pode se conectar a diversos outros. Novamente, para esta situação, o uso de coordenação baseada em eventos representa uma solução adequada, já que leva em consideração este tipo de comunicação coletiva.

Esta comparação dá a idéia geral por trás da proposta de modelo de programação em questão, porém, para que seja possível torná-lo funcional, é preciso que se resolvam alguns problemas. Primeiramente, é preciso estabelecer um nível, ou complexidade, mínimo para a tarefa desempenhada pela unidade de processamento. Esta complexidade precisa ser suficiente para justificar o overhead inerente ao uso de threads ou processos para sua execução e para que a latência do meio de comunicação não afete o desempenho a ponto de uma tarefa passar mais tempo comunicando-se do que executando seu processamento. Para isto, cabe lembrar que as implementações tradicionais de redes neurais artificiais são gerenciadas por um único processo que manipula e simula a propagação de estímulos entre os neurônios artificiais. Não costuma haver paralelismo real neste tipo de sistema exceto por aquele garantido pelo hardware em nível de instrução.

Por outro lado, trazer a uma tarefa mais responsabilidade, mas ainda manter-se fiel à proposta conexionista, onde estímulos são propagados de uma unidade de

execução às próximas, traz a necessidade de cada conexão carregar informações mais complexas e para isto é preciso estabelecer um protocolo de comunicação adequado bem como um modelo de tarefa que atenda a todos os aspectos do conexionismo.

5.1 Fundamentação teórica

De acordo com a abordagem do Processamento Paralelo Distribuído, em um modelo conexionista deve existir:

- Um conjunto de unidades de processamento;
- Um estado de ativação;
- Uma função de saída para cada unidade;
- Um padrão de conectividade entre as unidades;
- Uma regra de propagação para propagar padrões de atividades pela rede de conexões;
- Uma regra de ativação para combinar as entradas e produzir um novo nível de ativação da unidade;
- Uma regra de aprendizado pela qual os padrões de conectividade são alterados em função da experiência;
- Um ambiente onde o sistema deve operar;

A seguir é detalhado cada um destes aspectos bem como a maneira como foram tratados no modelo de programação proposto.

5.1.1 Unidades de processamento

Unidades de processamento são recipientes físicos ou lógicos para algoritmos que realizam alguma tarefa. Em um sistema conexionista, o algoritmo executado pelas unidades de processamento precisa ser tão simples quanto possível, já que o conhecimento do sistema está nas conexões e não no algoritmo em si.

No modelo em questão, as unidades de processamento passam a ser as UEs, substituindo a visão anterior onde a unidade de execução era uma thread ou um processo. Neste caso, portanto a UE é uma abstração que representa um conjunto de neurônios em operação. A estas UEs será dado o nome de Tarefa Conexionista ou simplesmente TC, e ao modelo que as define, MTC.

TCs permitem ao programador definir a quem elas serão conectadas bem como, a maneira para transformar um estímulo recebido em um novo estímulo a ser propagado pela rede de TCs. Elas definem, ainda, quais os tipos de estímulos que podem ser recebidos, como será explicado mais adiante neste capítulo.

O diagrama apresentado na figura 5.1 mostra uma TC com seus componentes. *T1*, *T2* e *T3* são os tipos de entradas aceitas. Os valores net_{Tn} são as entradas de tipo *Tn* recebidas em um dado momento. O valor *a* é o estado de ativação. O valor *o* é a saída gerada a partir de um estado de ativação *a*.

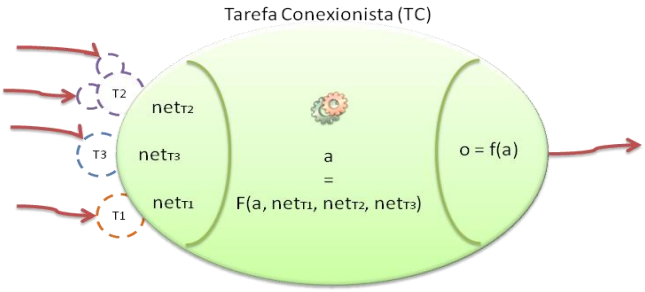


Figura 5.1: Diagrama conceitual de uma Tarefa Conexionista.

Internamente ao modelo, um mecanismo baseado no padrão *Master/Worker* é responsável por executar as TCs. Há um gerenciador global (*Master*) que atribui TCs a *threads* disponíveis em um pool (*Workers*) de forma que elas sejam executadas de acordo com a necessidade do sistema. Esta flexibilidade é importante, pois cada TC desempenha uma tarefa específica no sistema e não é necessário que todas estejam executando a todo o momento. Diminuindo o número de tarefas em execução, a carga do sistema para seu gerenciamento também diminui.

Cada TC realiza seu processamento de forma independente, sendo este paralelizado sempre que possível pelo escalonador do sistema operacional de acordo com a disponibilidade do hardware em que o sistema está executando.

5.1.2 Estado de Ativação

O estado de ativação, ou nível de ativação, de uma unidade de processamento é a informação que define o que a unidade de processamento realizou até um determinado instante. No modelo TC, o estado de ativação é representado pelo conjunto de variáveis de instancia de cada TC em execução. Cada uma pode possuir um conjunto distinto de variáveis que dizem respeito ao processamento que desenvolve. É em função do seu estado de ativação, juntamente com os estímulos de entrada, que uma TC irá produzir seus resultados. Com base nesta premissa, é correto afirmar que para um mesmo conjunto de estímulos de entrada fornecido em dois instantes distintos, a saída produzida pode ser diferente caso o estado de ativação seja diferente em cada um destes instantes.

5.1.3 Funções de Saída

Cada unidade de processamento interage com as demais através dos valores gerados por sua função de saída, que mapeia o estado de ativação a um sinal de saída a ser transmitido às unidades vizinhas. Em alguns casos, o sinal de saída corresponde exatamente ao nível de ativação, de forma que a função de saída corresponde à função identidade.

Em um sistema baseado no MTC, o estado de ativação precisa ser traduzido para um sinal de saída. O sinal de saída de uma TC é constituído por: um rótulo, que define sua identidade; um campo de carga útil, composto por um tipo complexo que pode conter qualquer informação relevante e um valor real, que representa a intensidade deste sinal e é essencial para as rotinas de treinamento e aprendizado,

atuando de forma idêntica ao sinal de entrada presente em uma rede neural tradicional

Como uma TC representa um conjunto de neurônios, ela pode transmitir diversos tipos diferentes de estímulos, porém o receptor pode estar interessado em apenas um subconjunto destes. Neste caso, cabe ao receptor definir quais os tipos de estímulos que serão aceitos através do seu padrão de conectividade.

Para simplificar o entendimento da comunicação entre TCs, a notação que representa um estímulo, é um par ordenado (x, i_x), onde x é um dado de qualquer natureza, podendo inclusive ser de um tipo complexo (composto por mais de um dado de diferentes tipos), i_x representa a intensidade do sinal de entrada para o dado x.

5.1.4 Padrão de Conectividade

As unidades de processamento precisam estar conectadas para que a interação entre elas seja possível. O padrão de conectividade corresponde à topologia da rede de conexões e é a característica que define o que o sistema, como um todo, conhece, e determina como ele responde a uma determinada entrada. Em um modelo tradicional de redes neurais, este padrão é mantido em uma matriz que determina os pesos de uma conexão entre um neurônio e outro.

No MTC, o armazenamento do padrão de conectividade é distribuído entre as TCs, de forma que cada uma mantém a lista de tarefas à qual está conectada, e estas, por sua vez definem como tratar uma entrada recebida por aquela conexão. São permitidas conexões em qualquer direção e mesmo conexões cíclicas. O diagrama apresentado na figura 5.2 representa as conexões entre diversas TCs. As setas indicam a direção do fluxo de sinais de saída. Os tipos de estímulos *Tn* são os tipos de estímulos que influenciam uma determinada TC.

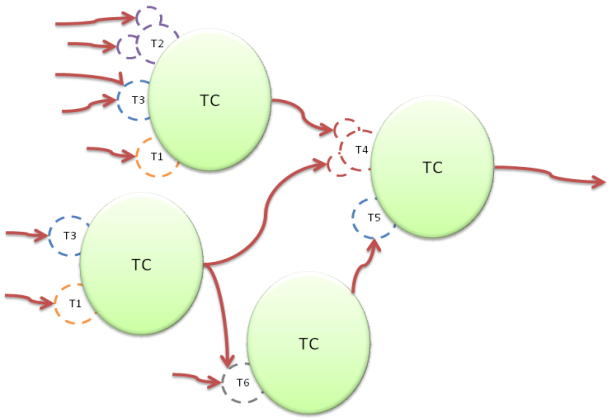


Figura 5.2: Diagrama do padrão de conectividade entre quatro TCs.

Existem duas formas para trabalhar com o padrão de conectividade das tarefas. Uma das formas é permitir que as TCs gerenciem as suas interconexões em tempo de execução. Isto torna a topologia fluida, onde é a ser possível alterar as interações entre as entidades dinamicamente. Desta maneira ajustes podem ser feitos para tornar as interações mais eficazes ou eventualmente permitir que o sistema aprenda maneiras novas de tratar os dados de entrada gerando saídas diferentes do que aquelas que foram previamente estabelecidas quando definida a topologia inicial.

Outra maneira de trabalhar com a topologia das conexões entre as tarefas é definir forma da rede de conexões programaticamente, e mantê-la imutável durante a execução do sistema. Para esta abordagem, é necessário que o sistema seja desenvolvido considerando todas as interações necessárias entre as TCs.

5.1.5 Regra de Propagação

É preciso estabelecer uma regra que receba a saída de uma unidade de processamento e a combine com o padrão de conectividade para produzir uma entrada para outra unidade de processamento. Em sistemas conexionistas onde apenas conexões excitatórias e inibitórias estão presentes esta regra costuma ser a soma aritmética dos valores de saída recebidos. Em sistemas mais complexos é necessária uma regra para cada tipo de conexão presente.

No MTC, a regra aplicada é bastante simples. Os tipos de estímulos aceitos por uma tarefa fazem parte da definição dela. Estes podem ser simples ou compostos. Um tipo composto é o conjunto de dois ou mais tipos simples. Nesta categoria, o tipo composto sincroniza o recebimento dos seus componentes e os encaminha para execução apenas quando todos os seus componentes forem recebidos. As entradas de uma TC são, portanto, os estímulos recebidos sem nenhuma transformação adicional. A figura 5.3 apresenta a definição dos tipos de estímulos aceitos por uma determinada TC. A figura 5.4 mostra como um estímulo composto ordena os estímulos simples que o compõe antes de permitir que ele ative a TC. No momento A, dois estímulos m_1 e m_2, de tipos diferentes estão sendo recebidos. A entrada do tipo *T1* ativa imediatamente a tarefa e seu processamento produz um sinal m_3 que é propagado para as suas vizinhas. O estímulo do tipo *T2* não é suficiente para ativar a tarefa, ficando acumulado até que, no instante B um novo estímulo do tipo *T2* é recebido, e então a thread processa ambos para produzir uma nova saída.

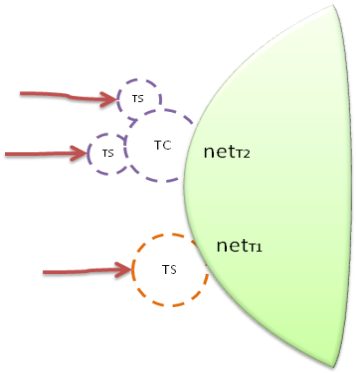


Figura 5.3: Tipos simples (*TS*) e tipos compostos (*TC*) de ativação.

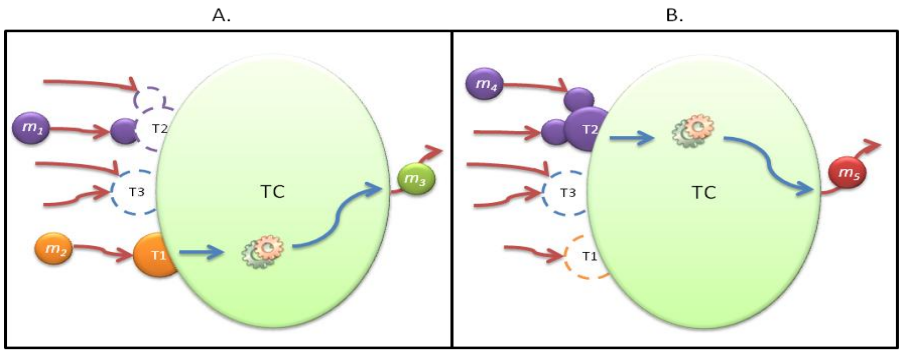


Figura 5.4: Diagrama contendo dois momentos na execução de uma TC.

5.1.6 Regra de Ativação

A regra de ativação consiste na função que processa as entradas para a produção de um novo estado de ativação.

Para o MTC, esta regra é definida pelo programador e individual a cada tarefa, ou seja, cada TC possui, como parte de sua definição, um algoritmo que processa as entradas e monta um novo estado de ativação em função delas.

5.1.7 Regra de Aprendizado

A regra de aprendizado em um sistema conexionista consiste na modificação do padrão de conectividade dinamicamente em função da experiência por ele realizada. Isto permite que o sistema como um todo modifique o que é conhecido por ele.

Esta regra, chamada de algoritmo de treinamento, costuma ser externa ao sistema, sendo coordenada por um mecanismo auxiliar que monitora todas as conexões entre as unidades de processamento e as ajusta para um determinado fim. Alguns algoritmos como o algoritmo de Hebb e *backpropagation* (HAYKIN, 1999) são bastante utilizados em redes neurais artificiais.

O MTC é tão flexível quanto uma rede neural artificial, e de maneira similar, a sua regra de aprendizado pode ser estabelecida de diversas formas. A diferença neste caso é que o modelo permite apenas que se enxerguem TCs, logo não é viável utilizar um mecanismo auxiliar, ou seja, o algoritmo de aprendizado deve ser gerido pelo próprio sistema. Por exemplo, é possível criar uma aplicação onde uma TC faça parte do padrão de conectividade da rede como um todo, e desempenhe a função de monitoramento e recompensa das demais. Neste caso, cada TC deve ser capaz de tratar estímulos de recompensa e realizar ajustes em suas regras de propagação.

5.1.8 Ambiente

Em um modelo baseado em PDP, é importante que esteja claro o ambiente para o qual o sistema existe, pois os padrões de entrada e saída podem fazer sentido somente para este ambiente. Por exemplo, não é esperado que uma rede neural treinada para reconhecer caracteres seja utilizada com sucesso para distinguir cores, pois se trata de dois ambientes distintos.

Enquanto no modelo PDP tradicional, as entidades intermediárias atuam como um sistema caixa-preta, onde não é possível identificar a função específica de cada

unidade de processamento, para o sistema TC, cada thread possui uma função conhecida, de forma que cada uma tem restrições quanto ao seu próprio ambiente. Estas restrições são satisfeitas pelo mecanismo de comunicação que atribui tipos às mensagens, permitindo que cada thread filtre apenas as mensagens que dizem respeito ao seu domínio.

5.2 Aprendizado e Treinamento

Por se tratar de uma das características mais interessantes e complexas de um sistema conexionista, esta seção inteira será dedicada ao mecanismo de aprendizado de TCs.

O mecanismo de aprendizado e treinamento proposto consiste em uma adaptação do algoritmo *backpropagation*, utilizado em redes neurais artificiais para treinamento de Perceptrons de Múltiplas Camadas. Neste algoritmo, a rede de conexões entre as unidades de processamento é treinada a partir de exemplos, e a técnica chamada de gradiente descendente é aplicada para realizar a atualização dos pesos de cada conexão entre as unidades até que o erro entre a saída desejada e a produzida seja reduzido a um patamar aceitável.

As próximas seções mostram o funcionamento deste algoritmo e a sua aplicação no MTC.

5.2.1 Algoritmo Backpropagation

O algoritmo *backpropagation* (que pode ser traduzido para o português como retroprograpagação) possui este nome devido à maneira como opera sobre a rede neural, pois o fluxo normal de uma rede neural é dos neurônios da camada de entrada em direção aos neurônios da camada de saída, mas no período de treinamento da rede, os erros, calculados pela diferença entre as saídas esperadas e obtidas, são trafegados da camada de saída em direção à camada de entrada.

A implementação deste mecanismo em uma rede neural artificial é feita através de um sistema auxiliar que atua em uma fase específica destinada ao treinamento da rede. Uma vez que os pesos sinápticos estejam ajustados, a fase de treinamento é encerrada e o algoritmo de treinamento é desativado. Se o treinamento for excessivo, a rede pode acabar memorizando os dados do conjunto de treinamento e perder sua capacidade de generalização. A este fenômeno dá se o nome “ajuste excessivo” (ou *overfitting*, em inglês), e evitá-lo exige ajuste nos parâmetros de treinamento como o volume de dados apresentados e o numero de vezes que estes dados serão repetidos à rede.

Dentro do modelo de programação proposto, grande parte das características do algoritmo *backpropagation* tradicional é utilizada, porém algumas mudanças tornam o mecanismo utilizado bastante peculiar.

Para ilustrar o funcionamento do algoritmo, e definir algumas equações que serão utilizadas mais adiante, é apresentado a seguir um resumo do material disponibilizado em (BERNACKI e WLODARCZYK, 2004).

Seja uma rede neural de três níveis (ou camadas) contendo duas entradas x_1 e x_2 e uma saída y como mostra a figura 5.5, onde cada círculo representa um neurônio artificial, e as setas representam as conexões entre estes neurônios.

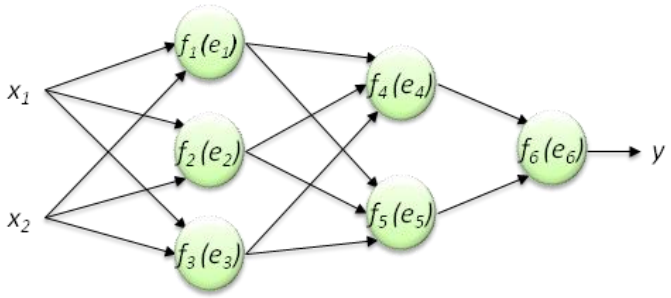


Figura 5.5: Rede neural de exemplo.

A figura 5.7 mostra cada neurônio decomposto em duas partes. A primeira é a regra propagação, que realiza o somatório dos produtos de coeficientes de peso pelos sinais de entrada do neurônio, definindo a entrada efetiva da rede *e* que respeita a fórmula da Figura 5.6:.

$$e = \sum_{j}^{n} w_j x_j$$

Figura 5.6: Valor de entrada efetivo da rede.

A segunda parte é a função de ativação que executa uma equação de ativação não linear $y = f(e)$, que neste caso é, também, o sinal de saída do neurônio.

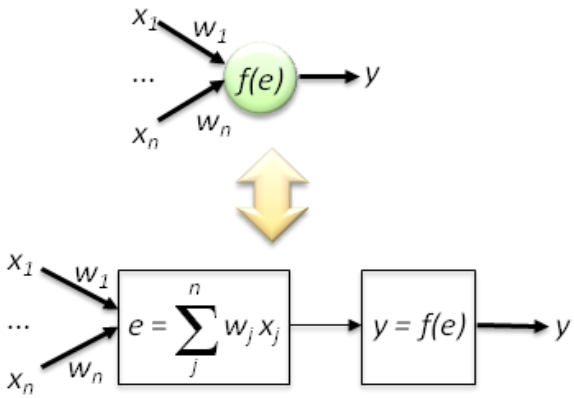


Figura 5.7: As duas partes de um neurônio artificial.

Para treinar cada neurônio dentro da rede neural, é preciso um conjunto de dados de treinamento. O conjunto de dados de treinamento consiste de sinais de entrada (x_1 e x_2) vinculados a seus valores de saida correspondentes (saída desejada) *z*.

O treinamento da rede é um processo iterativo. Em cada iteração os coeficientes de peso dos nós são modificados usando dados obtidos a partir do conjunto de treinamento. A modificação é calculada usando os passos descritos a seguir.

5.2.1.1 *Passo 1: Propagação dos Sinais de Entrada*

Cada iteração do treinamento inicia fornecendo-se dados de entrada provenientes do conjunto de treinamento e propagando-os através da rede. Após este estágio, é possível determinar os valores de saida para cada neuronio em cada nivel da rede.

A figura 5.8 mostra como o sinal é propagado através da rede produzindo sinais de saída para cada neurônio. Os símbolos $w_{(xm)n}$ representam os pesos das conexões de entrada em um neurônio n. O símbolo x_m é o valor fornecido como entrada e y_n é o sinal de saída do neurônio n.

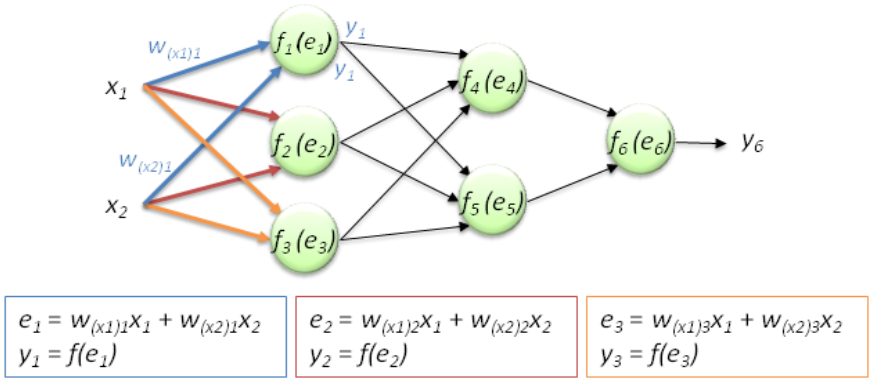


Figura 5.8: Propagação dos sinais pela camada de entrada da rede.

A propagação dos sinais pelo segundo nível da rede é representada na figura 5.9. Os símbolos w_{mn} correspondem aos pesos das conexões entre a saída do neurônio m e a entrada do neurônio n no nível seguinte.

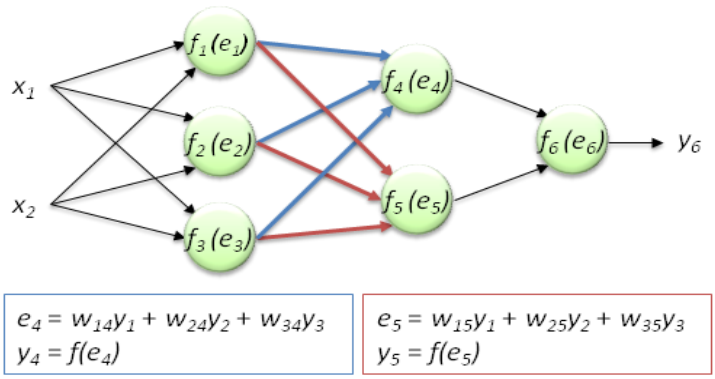


Figura 5.9: Propagação dos sinais pela camada intermediária da rede.

Por fim, os sinais chegam à camada de saída, como mostrado na figura 5.10.

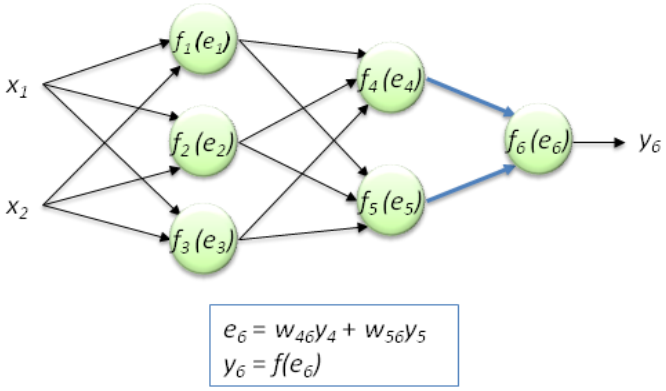


Figura 5.10: Propagação dos sinais pela camada de saída da rede.

5.2.1.2 Passo 2: Retro-propagação do Sinal de Erro

O próximo passo do algoritmo é comparar a saída *y* produzida pela rede com a saída desejada *z*. A diferença entre elas é chamada de sinal de erro δ do neurônio de saída e é calculado pela fórmula 5.1.

$$\delta = z - y$$

Figura 5.11: Cálculo do sinal de erro.

A figura 5.11 mostra a computação do erro de saída da rede e sua atribuição ao neurônio f_6.

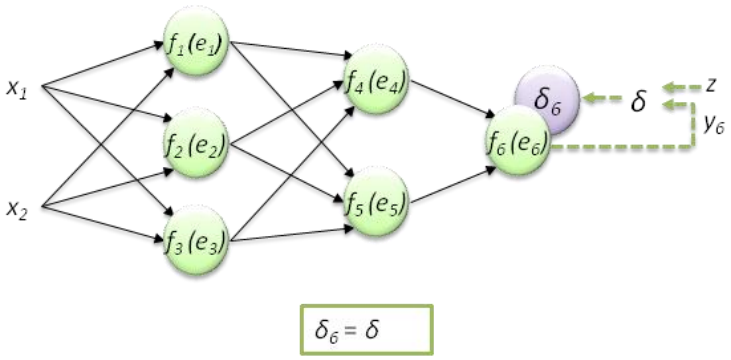


Figura 5.12: Propagação do sinal de erro no nível de saída da rede.

Não é possível computar o sinal de erro para os neurônios internos diretamente, porque o valor de saída desejada para estes é desconhecido. No entanto é possível propagar o sinal δ de volta para os neurônios internos, percorrendo a rede de maneira inversa. O cálculo do sinal de erro para os neurônios internos é feito pela fórmula 5.2 onde δ_n é o sinal de erro para o neurônio *n*. O símbolo *m* representa as conexões que partem do neurônio *n* em direção a qualquer outro neurônio.

$$\delta_n = \sum_{j}^{m} w_{nj}\,\delta_j$$

Figura 5.13: Cálculo do sinal de erro.

A figura 5.13 apresenta a computação dos sinais de erro δ_n para cada neurônio n. Os pesos w_{mn} são os mesmo utilizados no passo anterior para a propagação do sinal de entrada. Apenas a direção do fluxo de dados é diferente, indo das saídas para as entradas, de trás para frente (retro-propagação).

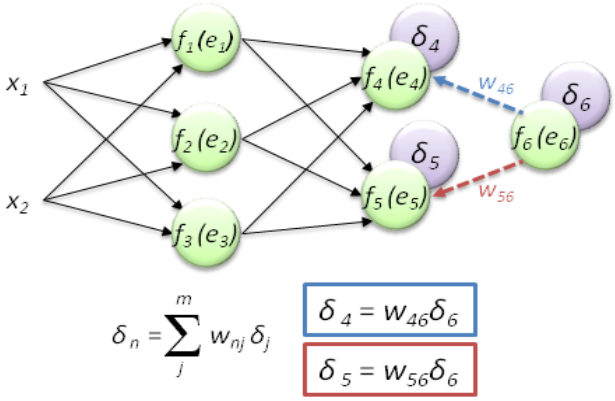


Figura 5.14: Propagação do sinal de erro no nível intermediário da rede.

Esta mesma técnica é usada para todos os níveis da rede. Quando o sinal de erro é recebido de mais de um neurônio, eles são somados como mostrado na figura 5.14.

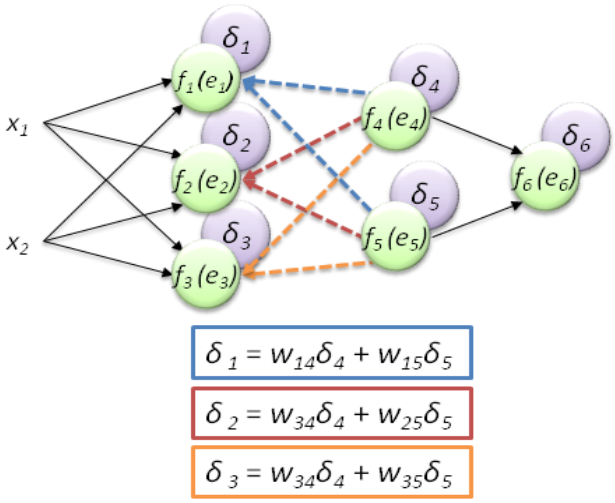


Figura 5.15: Propagação do sinal de erro no nível de entrada da rede.

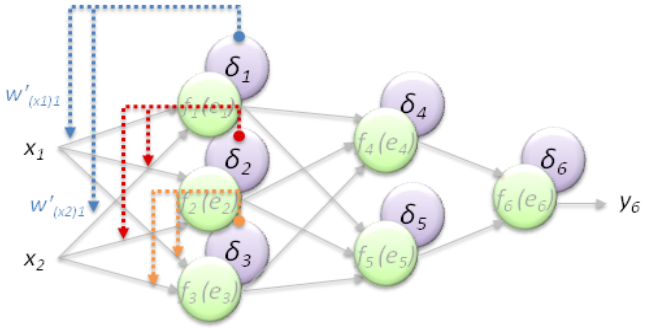
5.2.1.3 Passo 3:Atualização dos Pesos

Após o sinal de erro ter sido calculado para cada neurônio, os pesos para cada conexão de entrada podem ser atualizados através da fórmula da figura 5.16, onde w'_{mn} é o peso atualizado, $df_n(e_n)/de$ representa a derivada da função de ativação do neurônio *n*. O coeficiente η é a taxa de aprendizagem utilizada para influenciar a velocidade do treinamento.O símbolo y_m representa o valor de saída do neurônio *m* que serve como entrada para o neurônio *n*.

$$w'_{mn} = w_{mn} + \eta \delta_n \frac{df_n(e_n)}{de} y_m$$

Figura 5.16: Atualização do peso sináptico.

As figuras 5.17, 5.18 e 5.19 mostram, passo a passo, a atualização dos pesos da rede utilizada como exemplo.



$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e_1)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e_1)}{de} x_2$$

$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e_2)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e_2)}{de} x_2$$

$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e_3)}{de} x_1$$

$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e_3)}{de} x_2$$

Figura 5.17: Atualização dos pesos sinápticos no nível de entrada.

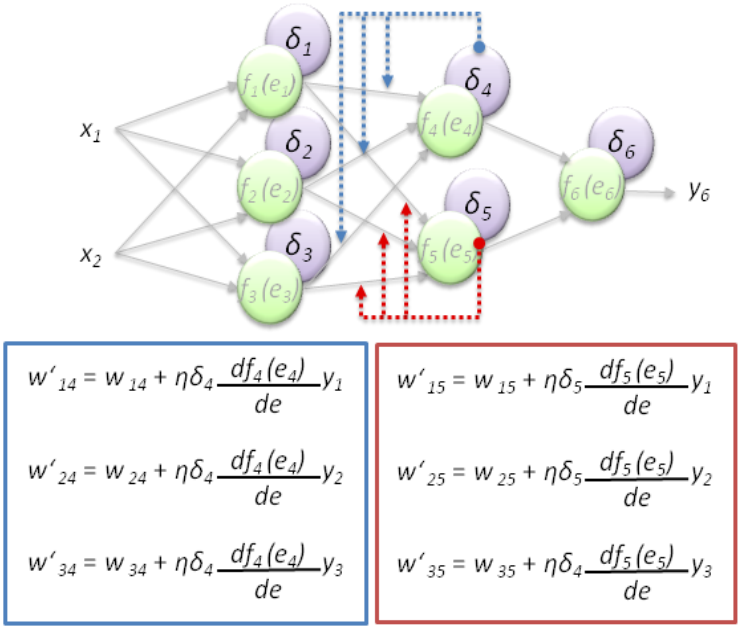


Figura 5.18: Atualização dos pesos sinápticos na camada intermediária da rede.

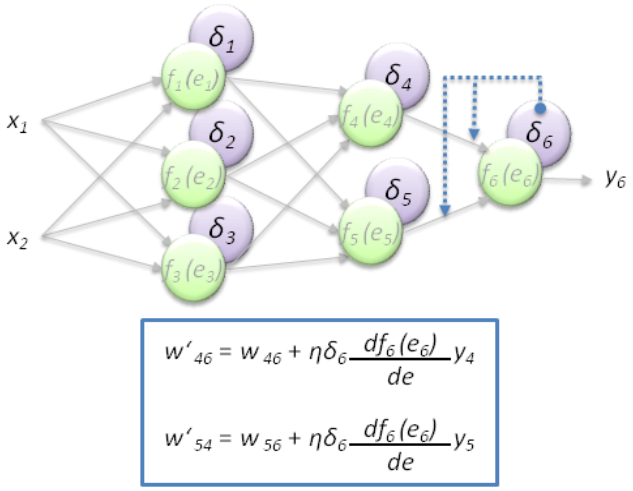


Figura 5.19: Atualização dos pesos sinápticos na camada de saída da rede.

5.2.2 Backpropagation no MTC

Para utilizar o algoritmo de treinamento backpropagation dentro do MTC, o conceito de cada passo, apresentado anteriormente, permanece inalterado; porem algumas mudanças fundamentais são necessárias para que o treinamento seja efetivo. As diferenças são apresentadas nas seções a seguir.

5.2.2.1 Pesos Sinápticos

Uma primeira diferença entre uma rede neural e um sistema baseado no MTC é a natureza dos dados trafegados pela rede. No primeiro caso, estes dados são uniformes, e consistem em geral, em números reais variando entre um curto intervalo (zero a um, por exemplo). Esta característica permite que estes dados sejam facilmente ponderados pelos pesos sinápticos dos neurônios subseqüentes

através de uma simples operação de multiplicação, como evidenciado pela fórmula de cálculo da entrada efetiva da rede mostrada na figura 5.6.

Dentro do MTC, dados complexos podem ser trafegados por meio de uma conexão. Estes dados, dentro da hipótese do modelo, representam as saídas de diversos neurônios ao mesmo tempo (ver 5.1.3), e, em função desta característica, uma conexão precisa possuir, também, diversos pesos. Para viabilizar a atribuição de pesos distintos dentro de uma mesma conexão, no MTC, os pesos são vinculados aos valores dos dados de entrada da conexão, de forma que valores diferentes possuam pesos independentes. Por exemplo, uma TC que recebe caracteres alfanuméricos de sua vizinha, pode atribuir um peso maior para o caractere 'A' do que para o caractere 'B', de acordo com sua necessidade e função.

5.2.2.2 Erro de saída

Duas etapas importantes para a execução do algoritmo de descida de gradiente são a identificação do erro de saída e a comunicação deste erro ao nível anterior de unidades de processamento.

No MTC, de forma diferente ao algoritmo backpropagation tradicional, a comunicação do erro de saída entre unidades vizinhas não é feito através de um mecanismo auxiliar, e nem exige uma fase de treinamento definida. Neste modelo, o erro é propagado sempre que for possível detectá-lo, permitindo o treinamento ininterrupto da rede por todo o período em que estiver em funcionamento, gerando um fluxo permanente de dados de treinamento em sentido contrário ao da propagação de sinais de entrada. Este fluxo entre as unidades da rede para propagação do erro de saída é realizado através de um espaço compartilhado de memória entre duas unidades. Uma o utiliza para escrever seu erro de saída e a outra para lê-lo. Este mecanismo de comunicação parte de uma hipótese inspirada na biologia, que considera que, em uma sinapse nervosa, a região de contato entre neurônios é utilizada para ajuste dos mesmos, caracterizando uma troca de informação, mesmo que indireta.

A figura 5.20 representa uma sinapse entre dois neurônios e o esquema lógico utilizado para representá-la dentro do MTC. Neste esquema lógico é definido um espaço de memória para compartilhamento de informação que é utilizado para comunicação do erro de saída às unidades vizinhas.

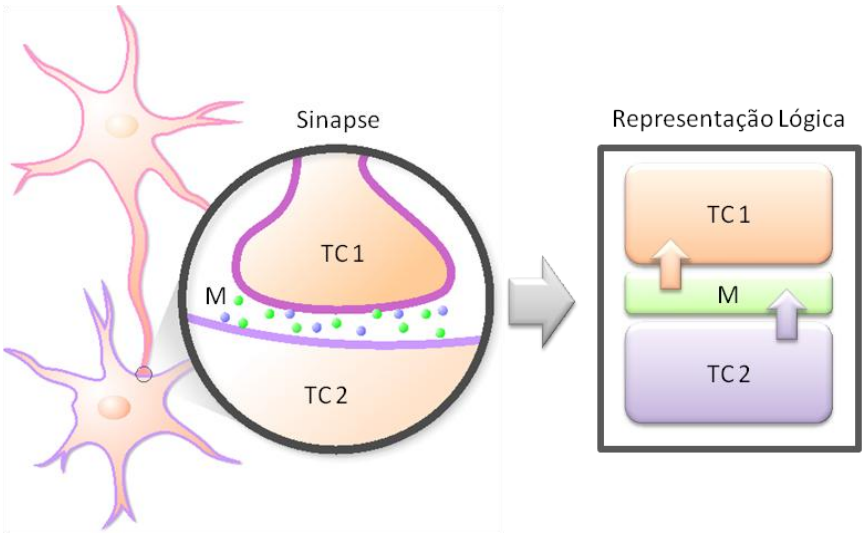


Figura 5.20: Diagrama representando a uma sinapse entre dois neurônios e a representação lógica desta sinapse no MTC. TC 1 e TC 2 são duas tarefas que utilizam o espaço M para a comunicação do erro de saída.

Dentro do MTC, existem duas maneiras para se obter o erro de saída de uma TC específica. Uma maneira é aquela utilizada pelo algoritmo backpropagation normal, onde uma unidade recebe um sinal de erro da sua vizinha à frente. Neste caso, o valor do erro recebido por uma TC, corresponde à fração com que esta influencia o erro da unidade à sua frente, e assim sucessivamente, sendo possível estabelecer uma cadeia de propagação de retorno, em múltiplas camadas. Esta fração é determinada pela formula apresentada na figura 5.16.

Outra forma para definir o sinal de erro, consiste em realizar o cálculo do erro de saída em função do conhecimento intrínseco da unidade em questão. Cada TC pode ser programada de maneira a conhecer a relação entre uma saída produzida e uma saída desejada para um determinado conjunto de entrada e a partir disto determinar um valor para o sinal de erro. O sinal de erro calculado desta maneira é necessário para dar inicio a um fluxo de aprendizado de toda a rede ou de uma parte dela.

5.2.3 Atualização dos pesos sinápticos

A atualização dos pesos de cada conexão é realizada utilizando o algoritmo de descida de gradiente em função do erro calculado para uma saída. A execução deste algoritmo dentro do MTC possui algumas peculiaridades.

Uma vez que o treinamento está sempre ativo, ou seja, não ocorre em uma fase específica, existe a possibilidade de ocorrer ajuste excessivo da rede (ver 5.2.1) por excesso de treinamento. Isto é evitado pelo mecanismo de detecção de erro, que solicita ajustes nos pesos apenas quando um dado apresentado for relevante o suficiente para que seja considerado por uma TC.

A definição do que é relevante a cada TC considera, basicamente, a quantidade de vezes que uma informação é apresentada em seqüência. Desta forma, se uma informação for apresentada repetidamente à rede, ela irá tentar aprender a sua relação com os dados de saída, porem se os dados forem apresentados sem repetição, os pesos permanecem inalterados. Este mecanismo funciona de forma automática pela própria dinâmica de propagação de erros entre as TCs.

Como definido na seção 5.2.2.2, há um espaço de memória entre duas TCs que é utilizado para comunicação do erro de saída entre elas. Este espaço de armazenamento mantém apenas o erro de saída correspondente à sua ultima ativação e nada mais. Porém, para atualizar os pesos sinápticos, o algoritmo de treinamento por descida de gradiente requer, além do erro de saída, os sinais de entrada que produziram aquela saída. Como o sinal de erro faz parte de um fluxo de dados distinto daquele que utilizado pela propagação de sinais de entrada e saída, no momento em que ele é recebido os valores de entrada que o produziram já não

são mais conhecidos. Sendo assim, para que uma TC possa ajustar seus pesos, é necessário que receba novamente dados de entrada que originaram a saída para a qual o sinal de erro é conhecido, e uma vez que estes forem recebidos, os pesos sofrerão um ajuste, pois a regra de aprendizado faz parte do fluxo de entrada e saída da rede.

Devido a esta característica, sempre que uma rede de TCs for submetida a dados semelhantes repetidamente, seus pesos serão ajustados e, em conseqüência disto, ela será treinada. Porém, se não houver repetição, não haverá ajuste nos pesos sinápticos e a rede permanecerá inalterada tornando desnecessária uma fase de treinamento, uma vez que a rede está sempre sujeita a ajustes.

Os passos para a atualização dos pesos de uma conexão em uma TC são apresentados a seguir em uma seqüência de diagramas. A notação utilizada para representar os dados de entrada e saída é aquela apresentada na seção 5.1.3.

A figura 5.21 representa uma TC com apenas uma conexão de entrada e uma conexão de saída no momento em que recebe um sinal de erro δ_n, onde n é um dado de saída de tipo complexo. Nesta TC, o sinal de erro ficará disponível à TC até que seja utilizado para seu treinamento ou sobrescrito por uma TC vizinha que propaga um novo sinal de erro.

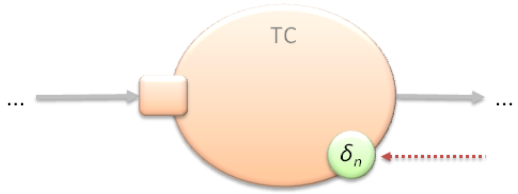


Figura 5.21: TC recebendo um sinal de erro para a saída n.

Uma vez que o erro para a saída de valor n é conhecido, na próxima ocasião em que este valor for gerado, os pesos das conexões de entrada que o produziram podem ser atualizados.

Esta situação é mostrada na figura 5.22 onde o valor n é produzido e seu erro δ_n é conhecido. Os dados de entrada (x, i_x) representam um conjunto de informações de entrada, onde x é um dado de tipo complexo e i_x é a intensidade do sinal de entrada para este dado. O símbolo w_x corresponde ao peso da conexão para o dado x.

Nesta mesma figura, é representado o cálculo dos valores e_n e i_n que representam, respectivamente, a entrada efetiva e a intensidade do sinal de saída. A entrada efetiva é calculada pela formula da figura 5.6.

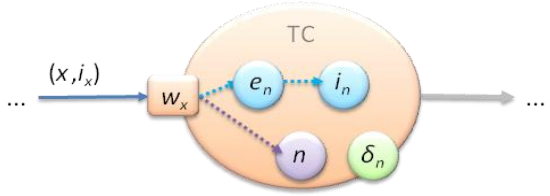


Figura 5.22: Dado n sendo produzido através da entrada (x, i_x).

A figura 5.23 apresenta o momento seguinte, onde dois passos importantes são realizados: o peso w_x é atualizado e o erro δ_x é calculado e propagado para a TC onde o dado x foi produzido e poderá, então, ser utilizado para o treinamento desta.

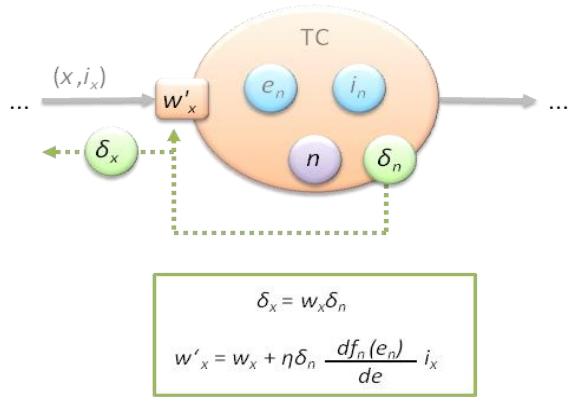


Figura 5.23: Atualização do peso da conexão entre duas TCs para um determinado dada (x) e a propagação do erro para a sua vizinha.

Uma vez que as regras de aprendizado são executadas, o fluxo normal de propagação é retomado e a saída (n, i_n) é produzida como mostrado na figura 5.24.

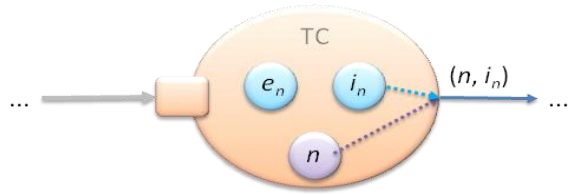


Figura 5.24: Continuação do fluxo de propagação de sinais.

Embora o exemplo apresentado considere apenas uma conexão de entrada e uma de saída, as mesmas regras podem ser aplicadas para múltiplas entradas e saídas.

Uma característica desta técnica de treinamento é o aproveitamento do paralelismo inerente ao modelo de programação, pelo fato da propagação de erro, e da atualização dos pesos entre as unidades, ser feito de maneira independente por cada TC.

5.2.4 Rede Resultante

A rede resultante possui algumas qualidades importantes. Primeiramente, o aprendizado é distribuído entre as unidades sendo, portanto, paralelizável. Além disto, o algoritmo faz com que a rede aprenda pela simples repetição dos dados de entrada, não exigindo, portanto, uma fase de treinamento, embora seja necessária a existência de uma unidade supervisora que conheça algumas das saídas desejadas para determinados conjuntos de entradas para que seja permitido o calculo do erro de saída.

A rede resultante possui características mistas entre uma rede neural tradicional e uma aplicação comum, onde o conhecimento de cada unidade de processamento é

inteligível, porem, por meio dos mecanismos de treinamento, a rede pode aprender a associar a produção de cada unidade para a produção de um valor adequado de saída.

5.3 Mecanismos de Implementação

O modelo de programação paralela proposto neste trabalho consiste em uma estrutura de algoritmo e uma série de mecanismos de implementação que tem por objetivo fazer com que o programador veja o sistema computacional como um conjunto de TCs e dessa maneira, decomponha o problema de forma a ser executado nelas.

Até agora foi apresentado o conceito por trás do modelo de programação paralela proposto e para isto a estrutura do algoritmo foi estabelecida com base no padrão de Coordenação Baseada em Eventos. Nesta seção, serão apresentados os mecanismos disponibilizados para transformar o algoritmo em um programa executável.

Para desenvolvimento destes mecanismos foi escolhido o paradigma de programação orientada a objetos, e a utilização da linguagem Java, que possui mecanismos nativos de suporte à programação paralela, sendo uma interface extremamente simples para desenvolvimento de aplicações orientadas a computadores com memória compartilhada baseados em múltiplos núcleos, e ao mesmo tempo, proporcionando um nível alto de abstração ao ambiente de hardware, garantindo portabilidade.

Por ser um modelo que utiliza orientação a objetos, as unidades funcionais são fornecidas pelo meio de classes. A seguir as principais classes e interfaces fornecidas em linguagem Java para este modelo serão detalhadas.

5.3.1 Tarefa Conexionista - classe StimuliEntity

Esta é a principal classe do modelo TC. Ela representa uma unidade de execução dentro do sistema.

Esta classe provê infra-estrutura para o recebimento de estímulos, seu processamento e a posterior produção de novos eventos.

De um ponto de vista do programador, a criação de uma TC passa pela extensão da classe StimuliEntity implementando os seguintes métodos:

- **activationRules:** este método corresponde às regras de ativação e é invocado sempre que um estímulo válido é recebido. Como parte do seu processamento, o estado de ativação deve ser atualizado;
- **activatesOnStimulusType:** este método é parte da regra de propagação e define quais são os estímulos válidos, ou seja, os que ativam esta TC;
- **outputFunction:** este método transforma o estado de ativação em estímulos de saída;

Em adição a estes métodos, cada TC precisa definir a quem está conectada. Para isto é usado o método **bindTo** que recebe como argumento uma TC, e deve ser invocado uma vez para cada TC receptora.

O uso do modelo requer que as atividades sejam divididas e colocadas dentro das TCs. Cada TC permite que o programador defina a lista de TCs que receberá os estímulos produzidos por ela e quais os tipos de estímulos que ela aceita. Além destes dois itens, é possível definir um conjunto de variáveis de instancia e uma função de ativação.

Uma vez que todos estes atributos estão definidos para todas as TCs, o sistema irá executar seguindo sempre uma mesma seqüência de passos, como mostrado na figura 5.25. Sempre que um estímulo for recebido, ele será validado. Caso seja um estímulo válido para a TC, ele será então passado à função de ativação que irá utilizá-lo para atualizar o seu estado de ativação.

A seguir a função de saída é chamada, produzindo novos estímulos com base no estado de ativação e os enviando às TCs interconectadas.

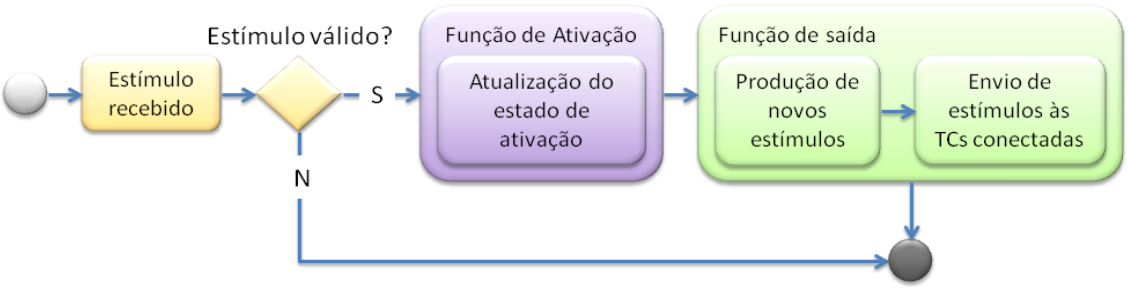


Figura 5.25: fluxo de trabalho de uma TC.

5.3.2 Sensores e Motores – interfaces StimulatorIfc e StimulableIfc

Para que o sistema se comunique com o mundo externo é necessário prover interfaces do mundo externo para o sistema e deste para fora. Para este fim existem dois tipos de interfaces StimulatorIfc e StimulableIfc. StimulatorIfc é uma interface que pode ser implementada em qualquer classe e permite esta classe gere estímulos de entrada ao sistema TC. StimulableIfc é o oposto, fornecendo a qualquer classe que a implemente, a habilidade de receber estímulos e fazer uso deles.

5.3.3 Comunicação

Cada receptor possui, em sua definição, uma lista com os tipos de mensagens que ele é capaz de processar (ver a seção 5.1.5). Sempre que uma mensagem é recebida, a TC é retirada do modo de espera e as mensagens relativas à lista são passadas ao método executor. Durante e após a execução, novas mensagens podem ser recebidas e ficarão enfileiradas até que a unidade de execução esteja livre novamente para tratá-las.

Por ser voltado a computadores multicore, este modelo define a comunicação entre TCs utilizando memória compartilhada. Para tanto, cada TC possui, internamente, uma fila compartilhada que coordena a produção e consumo dos estímulos. O modelo pode ser facilmente adaptado para máquinas com memória privada, bastando para isto modificar o mecanismo de transmissão de eventos para utilizar troca de mensagens.

5.3.4 Desempenho

As considerações quanto ao desempenho na utilização deste modelo são em essência as mesmas que se aplicam a um sistema que utiliza coordenação baseada em eventos. É preciso que se encontre concorrência de uma forma balanceada no sistema sendo modelado. Neste caso, a execução de suas funções de ativação irá possuir um tempo de execução adequado ao fluxo de dados recebido e também ao fluxo de dados esperado delas pelas TCs subseqüentes.

5.3.5 Escalabilidade

Para que o modelo de programação tenha boa escalabilidade e bom suporte a heterogeneidade é importante que o modelo de programação seja formulado para que possa aproveitar diferentes tipos de arquiteturas de hardware.

As arquiteturas de computadores pessoais multiprocessados da atualidade, são baseadas na arquitetura SMP, porém, com o aumento do número de processadores a opção por arquiteturas NUMA deve ocorrer.

Outra característica importante é a presença de infra-estrutura para SMT nos processadores, que também tem aumentado a sua presença devendo permanecer por um período considerável de tempo.

Para que o desempenho de uma aplicação não seja prejudicado por arquiteturas NUMA é necessário que ela leve em consideração a localidade dos dados manipulados, e neste intuito, o modelo de programação baseado em TCs permite que cada tarefa possua a sua fila de eventos localizada na memória do mesmo nodo que a executa.

Quanto ao uso de SMT, este só pode ser aproveitado se a aplicação possuir um balanceamento entre atividades com uso intenso de CPU e atividades com acesso a memória e E/S. O modelo baseado em TCs não oferece nenhuma facilidade ou empecilho ao aproveitamento de SMT, mas cabe ao programador desenvolver algoritmos que possuam este balanceamento.

5.4 Trabalhos Relacionados

A seguir são apresentados alguns trabalhos relacionados a este na área de programação paralela.

Há uma diversidade de trabalhos que apresentam a paralelização de modelos oriundos do PDP. No trabalho de pesquisa intitulado *Parallel Event-Driven Neural Network Simulations Using the Hodgkin-Huxley Neuron Model* (LOBB, CHAO, *et al.*, 2005) é demonstrado um modelo de redes neurais orientadas à execução paralela com utilização de coordenação baseada em eventos. A abordagem utilizada, no entanto é limitada a uma instancia do PDP, o modelo Hodgkin-Huxley com topologia feed-forward (sem retro-alimentação). A pesquisa apresenta formas para paralelizar a resolução das equações deste modelo utilizando uma biblioteca de suporte a concorrência.

Em *The Concurrent Graph: Basic Technology for Irregular Problems* (TAYLOR, WATTS, *et al.*, 1996) um modelo de programação paralela de alto desempenho é desenvolvido. Para utilizá-lo é disponibilizada uma biblioteca que oferece mecanismos para modelagem de um sistema paralelo e abstrai o método de

comunicação entre as tarefas e a arquitetura física do sistema computacional. Neste modelo, cada tarefa é representada por um nodo em um grafo. Cada nodo possui um estado, uma lista de comunicação e as rotinas especificas da aplicação. O estado é o conjunto de dados que representa a divisão do problema, a lista de comunicação representa as arestas do grafo e definem a direção das mensagens enviadas de um nodo para outro. As rotinas específicas são dependentes da aplicação e realizam processamento sobre o estado.

A biblioteca desenvolvida neste modelo oferece mecanismos para balanço dinâmico de carga e para computações adaptativas. Para este ultimo operações de divisão (*split*) e junção (*merge*) permitem que nodos sejam combinados ou separados dinamicamente, estabelecendo a topologia do grafo em tempo de execução. Isto é especialmente útil em computações com volume grande de dados, onde a divisão dos nodos separa os dados e permite que sejam operados em paralelo.

6 SISTEMA DE CONTROLE AUTÔNOMO PARA FUTEBOL DE ROBÔS

Neste capítulo será apresentada uma visão geral sobre o sistema de controle e sua função dentro do futebol de robôs. A seguir serão descritas as etapas realizadas para sua construção utilizando a linguagem de padrões para processamento paralelo apresentada no capítulo 3, incluindo a maneira como o paralelismo foi encontrado, a estrutura do algoritmo desenvolvido e as estruturas de suporte utilizadas.

Para este trabalho é considerado que existe uma divisão de responsabilidades entre um mecanismo motor, que envia comandos por rádio aos robôs, um mecanismo de visão, que transforma gráficos em um modelo lógico do jogo, e um sistema de controle autônomo, que faz com que os robôs joguem futebol. Sendo assim, a tarefa do sistema de controle, como seu nome sugere, é comandar os robôs para que eles atuem como um time de forma autônoma.

Dentro da liga SSL da Robocup, este sistema reside em um computador dedicado e fica situado entre o sistema de visão e os robôs físicos. É seu papel, portanto, transformar dados de entrada, fornecidos pelo sistema de visão, em comandos de saída, para movimentar os robôs. O diagrama da figura 6.1 apresenta uma visão simplificada de um time da liga SSL. O time de robôs é monitorado por câmeras montadas sobre o campo. Os dados são passados das câmeras para um sistema de controle que irá processar estes dados analisando o jogo e produzindo comandos, transmitidos por meio de rádio para serem executados pelos robôs.

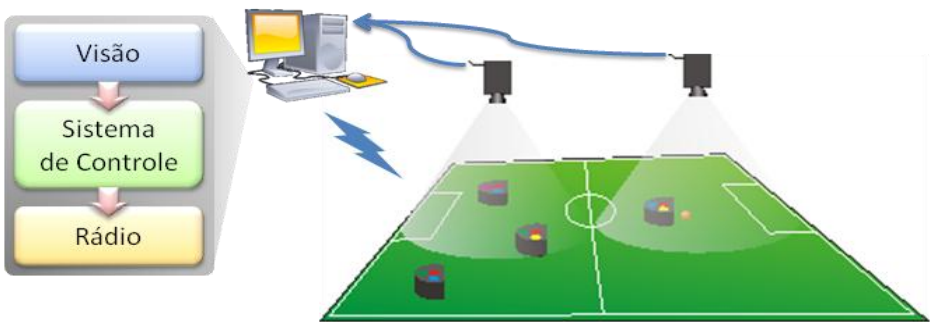


Figura 6.1: Funcionamento de um time da liga SSL.

Os dados provenientes do sistema de visão são utilizados para produzir um modelo lógico do jogo de futebol que considera a identidade, posição, velocidade linear e angular de cada robô e velocidade da bola. Alguns dados são obtidos por meio de respostas de rádio diretamente dos robôs, como, por exemplo, o estado do

chutador e do driblador, e desta maneira o sistema de controle recebe, como entrada, uma visão quase completa do ambiente em que pode atuar.

Os comandos aos quais os robôs respondem são, vetores de velocidade, que indicam a direção, sentido e intensidade com que o robô deve se deslocar; vetores de velocidade de rotação, com função de definir sentido e intensidade da rotação dos robôs; intensidade do driblador e intensidade do chutador.

O sistema de controle autônomo precisa ser capaz de avaliar a situação atual do jogo e definir, através dessa situação e dos seus próprios mecanismos internos, quais comandos devem ser passados aos jogadores, existindo uma infinidade de formas para tornar isto realidade. A maneira mais tradicional consiste em produzir um banco de dados de jogadas utilizando algoritmos para cálculo de trajetórias, condução da bola, chutes, etc. Este banco de jogadas é mapeado a situações possíveis em um jogo, e uma jogada é então selecionada de acordo com uma determinada situação. As jogadas, em geral possuem uma série de algoritmos, um para cada jogador em campo, de forma que cada jogador executa um algoritmo separado, de acordo com seu papel, também definido como parte da análise do cenário do jogo. Quanto maior o número de jogadas conhecidas, maior a autonomia do time e a sua capacidade de surpreender o oponente.

Tentando manter-se fiel à abordagem mais tradicional, o sistema de controle apresentado neste trabalho possui essencialmente esta mesma estrutura, procurando encontrar pontos em que ela pode ser paralelizada e definindo como o paralelismo pode ser explorado. As próximas seções irão demonstrar como este problema se parece ao ser submetido à linguagem de padrões descrita no capítulo 3 com as restrições impostas pelo modelo de programação do capítulo 4.

6.1 Encontrando Paralelismo

O primeiro passo para a modelagem de um programa paralelo, é a definição do que pode ser paralelizado. Não é eficaz, porém, iniciar esta busca sem conhecer claramente o sistema computacional onde o programa deve ser executado, pois, como discutido anteriormente, a plataforma de hardware impõe restrições sobre os canais de comunicação e o gerenciamento de recursos compartilhados que podem tornar o programa extremamente ineficiente se a escolha da estrutura paralela for inadequada.

Para o sistema de controle autônomo de um time da SSL da Robocup, o ambiente de hardware é razoavelmente bem definido, pois fica praticamente restrito a um computador pessoal. O ambiente de software precisa ser adequado a este hardware, mas é livre a cada equipe definir o que executa neste computador. Para este trabalho, será considerado que o ambiente de software utiliza um sistema operacional com suporte a multi-tarefas, como Windows ou Linux. A linguagem de programação utilizada é Java, adicionada da biblioteca de suporte ao modelo de programação do capítulo 5.

6.1.1 Decomposição de tarefas

O paralelismo que se deseja explorar é o de granularidade grossa, ou seja, *threads*. Para tanto, é preciso decompor o programa em partes suficientemente

independentes para que o overhead com comunicação e o desperdício de tempo com sincronização sejam amortizados pelo ganho de tempo de processamento obtido pela execução paralela destas partes.

O primeiro passo na busca por paralelismo é definir qual o tipo de paralelismo que pode ser explorado. O sistema de controle na liga SSL pode ser classificado como uma célula robótica inteligente. Neste tipo de sistema robótico não há uma quantidade maciça e regular de dados em que possa ser utilizado o paralelismo de dados como mecanismo principal, mas, este problema possui uma estrutura irregular e divisível de tarefas que pode ser paralelizada. A divisão hierárquica costuma seguir a forma descrita na figura 6.2 onde três níveis de controle existem: organização, coordenação e execução.

Figura 6.2: Estrutura funcional de uma célula robótica inteligente como definida em (JACAK, 1999).

As tarefas do nível de Organização são aquelas que analisam constantemente o estado do jogo e usam de sua base de conhecimento para definir o que deve ser feito pelo time e qual parte cabe a cada jogador. As tarefas do nível de Coordenação, por sua vez, definem como algo deve ser feito e as do nível de Execução o fazem.

A segunda etapa é definir quais tarefas pertencem a cada nível. Para a Organização, uma tarefa é responsável por definir a estratégia do jogo e outra por definir quais as características de cada jogador e dividí-los em grupos para que possam desenvolver atividades coletivas.

A Coordenação possui os seguintes tipos de tarefas. Para cada grupo de jogadores, uma tarefa é responsável por executar uma jogada. Para cada jogador dentro da jogada, uma tarefa é responsável por realizar uma atividade individual que contribua para a jogada e para cada atividade individual, uma série de tarefas executam ações independentes e componíveis, que contribuem para a realização da atividade. Sendo assim, existe uma diversidade grande de tipos de jogadas, atividades e ações e cada uma delas precisa ser implementada em uma tarefa específica.

O nível de execução é composto pelas tarefas que realizam a interface com os robôs enviando comandos a estes e monitorando seus estados.

6.1.2 Agrupando e Ordenando as Tarefas

A decomposição realizada anteriormente propõe uma hierarquia entre as responsabilidades de cada tarefa. Esta hierarquia já permite atingir dois grandes objetivos da modelagem do sistema: agrupar tarefas e ordená-las de acordo com suas dependências para facilitar o seu gerenciamento.

Além das tarefas citadas anteriormente ainda existe um ultimo nível hierárquico, que representa as interfaces do sistema com o mundo externo.

Sendo assim, neste trabalho foram produzidos cinco grupos de tarefas ordenados em níveis como mostrado na figura 6.3. Esta divisão é coerente com a célula robótica inteligente e constitui uma arquitetura robusta que pode ser utilizada em outras aplicações de controle de times de agentes.

Figura 6.3: Grupos ordenados de tarefas.

Esta ordem reflete fortemente a direção do fluxo de controle, já que as tarefas em níveis mais baixos só existem em função daquelas de níveis mais altos. A ordem define também, em boa parte, o fluxo de dados, pois o uso de coordenação baseada em eventos vincula dados ao controle.

Quanto mais alto o nível, maior a influência nos demais, e mais duráveis precisam ser seus resultados. Entenda-se que as alterações no estado imediato do jogo terão menos influência nos resultados quanto maior o nível em questão, de forma que níveis mais altos servem para dar estabilidade ao sistema e níveis mais baixos servem para dar agilidade ao time para respostas em tempo hábil.

O nível mais alto e, portanto, com maior responsabilidade perante o time, é o nível de Organização. Nele, o estado do jogo é avaliado para definir a estratégia a ser utilizada além das jogadas que precisam ser executadas que se alinham à estratégia e à situação do jogo. Estes níveis definem “o quê” precisa ser feito e “quando”.

Em segundo lugar está o nível de Coordenação onde as jogadas são produzidas. Neste nível o sistema define “como” as operações devem ocorrer.

Por fim o nível de Execução intermédia a interação entre a camada física e demais tarefas, tornando seus resultados visíveis.

Considerando o modelo de programação em uso, este é um bom momento para iniciar a avaliação do balanço de carga entre as tarefas de forma a evitar gargalos no tempo de transposição do caminho crítico do algoritmo. Tendo esta premissa em mãos, é preciso garantir que todas as tarefas que operam em um mesmo nível possuam uma complexidade semelhante.

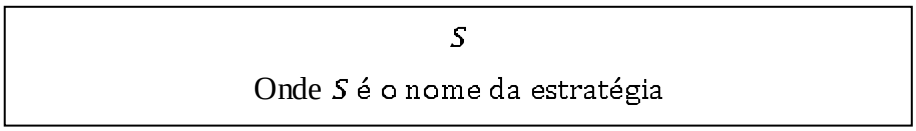
A seguir cada um destes grupos de tarefas é descrito em mais detalhe.

6.1.3 Estratégia

Estratégia é o nível que observa o jogo analisando o que passou e o que está por vir e definindo como o time deve se comportar. Aqui também é acompanhado o desempenho da camada de aglomeração de forma a validar a eficácia da estratégia escolhida. Este nível vai dizer aos restantes se o time deve atacar ou defender ou simplesmente se posicionar.

Uma vez determinada qual a estratégia a seguir, a responsabilidade é passada para os níveis inferiores. É importante que a decisão tomada seja suficientemente persistente para não invalidar todo o trabalho de criação de comandos motores realizado nos demais níveis. Para isto, a invalidação de uma definição de estratégia se vale de um sistema de retorno do nível de aglomeração, de modo que o progresso das jogadas sendo realizadas por cada grupo constitui um fator determinante para a continuação ou abandono de uma estratégia.

Para representar a estratégia será utilizada a notação:



6.1.4 Divisão de Grupos

Este nível recebe a estratégia do time e com base nesta, processa a posição dos robôs para gerar um ou mais grupos de jogadores. Em cada grupo, os elementos são avaliados por fatores como, a sua proximidade à bola e a posição relativa a pontos estratégicos do campo, e recebem um conjunto de características que servem para definir o papel que cada jogador vai desempenhar em uma jogada.

Uma vez definidos os grupos e as características dos jogadores, o sistema de aglomeração procura, entre todas as definições de jogadas conhecidas, por uma ou mais jogadas adequadas a cada grupo. Uma jogada é selecionada apenas quando seus requisitos são satisfeitos pelos grupos e seus componentes. Uma explicação detalhada sobre os requisitos de uma jogada são explicados na próxima seção. Por hora basta considerar que uma jogada estabelece quais são as características que os jogadores precisam ter para que ela seja executada de forma adequada. O nível de aglomeração tem acesso a estas necessidades e somente dispara uma jogada caso estas necessidades sejam satisfeitas.

Caso jogadas adequadas sejam encontradas, elas são iniciadas e passam a determinar o comportamento dos jogadores em um grupo. Se houver mais de uma jogada adequada para um grupo, é selecionada aquela com maior coeficiente de sucesso. Uma vez consolidados os grupos e selecionadas as jogadas, estas serão disparadas e monitoradas e neste ponto se encerra a responsabilidade do nível de Aglomeração.

De forma resumida, o nível de aglomeração mapeia uma estratégia a uma ou mais jogadas por meio do estado atual do jogo.

Para representar os grupos neste trabalho será usada uma notação semelhante à de conjuntos matemáticos:

$$G_n = \{R_{1_{\{C_1,\ldots,C_j\}}}, \ldots, R_{i_{\{\ldots\}}}\}$$

Onde G_n é o grupo de número n, R_m são os robôs pertencentes ao grupo, numerados de 1 a i cada um com seu conjunto de características, C_o é uma característica de um robô.

6.1.5 Jogadas

Enquanto os níveis superiores estão mais focados na tática, este nível já apresenta características operacionais. Jogadas são conjuntos de atividades que envolvem um ou mais jogadores com o objetivo de executar uma tática estabelecida pelos níveis superiores. A responsabilidade deste nível é acompanhar as atividades desempenhadas pelos níveis inferiores e dispará-las no momento adequado ao longo do tempo de forma a desempenharem um comportamento de conjunto. Há sempre uma jogada por grupo. Dependendo do número de grupos formados pode existir mais de uma jogada acontecendo ao mesmo tempo sendo que cada jogada é gerenciada individualmente por uma *thread* que dispara e monitora o sucesso das suas atividades.

Comparando o nível de Jogada e seu nível inferior (Atividade) descrito na seção 6.1.6, a noção de conjunto é a principal diferença. Enquanto a atividade foca em um único jogador, a jogada precisa fazer sentido para um grupo de jogadores.

Em uma troca de passes, por exemplo, um grupo de dois jogadores, a uma distância razoável, se deslocam na mesma direção. Um dos jogadores precisa se preparar para receber um passe, o outro precisa por sua vez, passar a bola. Este conjunto orquestrado de atividades é o que constitui uma jogada.

6.1.5.1 Configuração de Requisitos

Todas as jogadas possuem uma configuração de requisitos que permite ao nível de aglomeração saber quando ela é adequada a um grupo ou não.

A configuração de requisitos das jogadas é representada pela seguinte notação:

$$J[\aleph(G), \{C_1, \ldots, C_j\}, S]$$

Onde J é o nome da jogada, $\aleph(G)$ é a cardinalidade de um grupo G. O conjunto $\{C_1,\ldots, C_j\}$ representa as características necessárias à jogada e S é a estratégia à qual a jogada pertence.

6.1.6 Atividades

Atividade é o nível mais alto de controle individual de um robô. Este nível irá combinar e coordenar uma serie de ações que permitirão a um robô realizar um objetivo de curto prazo.

Uma atividade, sozinha não tem utilidade para o time como um todo, ela define a meta de um jogador, mas esta meta é apenas um meio para este jogador contribuir com a jogada sendo realizada pelo grupo.

Por sua vez, atividades orquestram ações acompanhando seu andamento e ajustando sua influencia nos movimentos do robô.

6.1.7 Ações

Ações correspondem ao nível mais baixo de controle, atuando individualmente em cada robô, traduzindo um desejo do sistema em um comando a ser enviado ao robô.

Mesmo para um único robô, são poucas as ações que tem utilidade quando tomadas individualmente. Ações são meios para se desempenhar uma atividade, logo para que sejam úteis precisam ser combinadas.

Não há restrições sobre os algoritmos utilizados para realizar cada ação, porém, seja qual for a ação, ela deve produzir comandos motores que possam ser executados por um robô.

Cada ação é executada individualmente por uma *thread*, mesmo quando faz parte de uma atividade em que há outras ações. Desta forma, seus resultados são produzidos de maneira independente. Já que é possível que um único robô esteja desempenhando mais de uma ação ao mesmo tempo, é necessário garantir que os comandos produzidos por uma ação não interfiram negativamente com os resultados de outra. A abordagem utilizada para evitar este problema foi estabelecer que cada ação possui responsabilidades diferentes; por exemplo, enquanto uma ação procura mover um robô a um alvo, outra procura evitar obstáculos, ao se compor estas ações, o robô vai para o seu alvo evitando as barreiras no meio do caminho.

Existem alguns casos especiais em que uma ação precisa conhecer o resultado de outra para que se comporte adequadamente. Neste caso é necessário que se estabeleça um canal de comunicação entre elas.

6.1.8 Movimento

Este é o nível responsável por condensar o resultado das ações, combiná-los e, por fim, comunicá-lo aos robôs, atuando como intermediário entre a execução das táticas e o meio físico.

Como visto anteriormente, as várias ações que executam em um mesmo momento para cada jogador, produzem resultados individuais de maneira ininterrupta; porém, estes resultados visam um objetivo comum. Para que os comandos resultantes das ações permitam ao robô se deslocar e atuar da maneira esperada, eles precisam ser combinados em um único comando.

Um problema que o paralelismo adiciona à arquitetura do sistema de controle, é a natureza assíncrona das ações. Graças a esta característica, é normal que os resultados produzidos por elas sejam disponibilizados em momentos diferentes. Para suprir à necessidade de compor estes resultados assíncronos em um único comando, o nível de movimento desempenha o papel de sincronizador, estabelecendo janelas de tempo onde vários comandos são recebidos do nível de ações, combinados em um único comando por robô e, por fim, enviados a cada um dos robôs.

É importante ressaltar que a necessidade de um ponto de sincronização existe de um ponto de vista da arquitetura proposta; porém, o tamanho da janela de tempo em que um comando deve aguardar para ser enviado é uma função da largura de banda do meio físico de comunicação com os robôs. Quanto mais rápido for este canal de comunicação, menor o intervalo entre o envio de um comando e o seguinte.

6.1.9 Leitura de Dados

Neste nível, os dados da visão são recebidos e estruturados em um modelo lógico do estado do jogo, podendo ser entendido como o nível sensorial do sistema.

A leitura de estado ocorre periodicamente, mas de forma tão rápida quanto suportada pelo sistema.

6.2 Estrutura do Algoritmo

O modelo de programação baseado em TC define a estrutura do algoritmo dentro do padrão de coordenação baseada em eventos. Já dentro da linha de pensamento do modelo TC, neste momento é definido o grafo de interconexões entre cada TC. Neste momento não estão sendo consideradas as tarefas individualmente, mas os grupos de tarefas, apenas para que se possa projetar as dependências de dados entre as TCs.

A figura 6.4 apresenta um esboço do gráfico de interconexões entre tarefas de diferentes grupos. Neste diagrama é possível verificar que tarefas trafegam dados em ambos os sentidos. Os dados trafegados de cima para baixo correspondem a ordens de comando, os de baixo para cima, atualizações de estado. A partir destes dois canais principais, uma TC é monitorada por sua superior e comanda suas inferiores.

Figura 6.4: Conexões entre TCs no sistema de controle autônomo.

Uma vez definida a estrutura de comunicação do algoritmo, é necessário definir os algoritmos de execução que correspondem às tarefas em cada nível.

Como definido anteriormente, existem alguns níveis de responsabilidade distintos dentro da aplicação de controle. Dentro de cada nível é necessário que sejam definidas as tarefas que o sistema sabe desempenhar. Estas tarefas são definidas por meio de um ou mais algoritmos de execução, que ao serem combinados determinam o que o sistema sabe fazer. Uma vez que estejam corretamente definidas, cada tarefa deve ser mapeada a uma TC e sua execução ocorrerá de forma paralela sempre que as suas dependências permitirem. Para provar o conceito utilizado neste trabalho, alguns algoritmos de execução foram escolhidos para implementação. A seguir estes algoritmos são descritos em mais detalhe, sendo apresentados de acordo com seu nível de responsabilidade.

6.2.1 Estratégia

Existe apenas uma tarefa responsável por definir a estratégia. Portanto apenas um algoritmo é definido para este nível. A estratégia é definida através de uma ponderação entre fatores críticos de sucesso tais como a posição da bola no campo, o risco de um contra ataque e a chance de sucesso de um ataque. A inferência da estratégia com base nestes fatores é realizada através de um sistema de lógica *fuzzy*.

As estratégias possíveis são:

- *Attack* que representa oportunidade para jogadas de ataque.
- *Defense* que representa oportunidade para jogadas de defesa.
- *Position* que indica que o time deve se posicionar taticamente ou para cobrar uma bola parada.

A seguir são apresentados detalhes sobre a síntese da estratégia.

6.2.1.1 Sistema de inferência de recomendação estratégica

A recomendação de estratégia é a análise feita sobre a situação do jogo, que indica se o time deve: atacar, defender-se ou simplesmente posicionar-se dentro do campo. Esta análise utiliza um mecanismo de inferência e algumas informações do jogo como subsídio para indução.

A inferência consiste em um sistema *fuzzy* de duas etapas. A primeira etapa é definida pelo bloco funcional descrito abaixo em notação IEC 61131-7 (ABBOTT e NEEMA, 1997). Algumas partes do código foram omitidas para facilitar a compreensão.

```
// Identify some coefficients to be used in the
strategy definition
FUNCTION_BLOCK strategiesInputs

    // Define input variables
    VAR_INPUT
        // distance from the ball to the opponent goal
        ballToOpponentGoalDistance: REAL;
        // distance from the ball to the own goal
        ballToOwnGoalDistance: REAL;
    END_VAR

    // Define output variable
    VAR_OUTPUT
     //Risk that an attack play will succeed
        attackRisk : REAL;
     //Risk that an counter attack play will succeed
        counterAttackRisk : REAL;
    END_VAR

    ...

    RULEBLOCK extractInputs
        RULE 1 : IF ballToOpponentGoalDistance IS low
                    THEN attackRisk IS low;

        RULE 2 : IF ballToOpponentGoalDistance IS
medium
                    THEN attackRisk IS medium;

        RULE 3 : IF ballToOpponentGoalDistance IS high
                    THEN attackRisk IS high;

        RULE 4 : IF ballToOwnGoalDistance IS low
                    THEN counterAttackRisk IS low;

        RULE 5 : IF ballToOwnGoalDistance IS medium
                    THEN counterAttackRisk IS medium;

        RULE 6 : IF ballToOwnGoalDistance IS high
                    THEN counterAttackRisk IS high;
    END_RULEBLOCK

END_FUNCTION_BLOCK
```

Esta primeira etapa recebe como parâmetros de entrada a distância entre a bola e o gol do oponente, normalizada no intervalo 0 – 10 e a distância entre a bola e o próprio gol, normalizada no intervalo 0 – 10. A partir destes parâmetros, são inferidos, como variáveis de saída, o risco de que uma jogada de ataque tenha sucesso e o risco de que um contra-ataque tenha sucesso.

A segunda etapa da inferência é definida pelo sistema:

```
// Block for attack strategy
FUNCTION_BLOCK strategies
    // Define input variables
    VAR_INPUT
        // Risk that an attack will succeed
        attackRisk : REAL;
        // Risk that a counter-attack will succeed
        counterAttackRisk : REAL;
        // Minor distance between a team-mate and the
ball
        ballDistance : REAL;
        // Minor distance between an opponent and the
ball
        ballToOpponentDistance : REAL;
        // The state of the game
        gameState : REAL;
    END_VAR
    // Define output variable
    VAR_OUTPUT
        //The recommendation on how the team shall
behave
        strategyRecommendation : REAL;
    END_VAR

    ...

    RULEBLOCK defineStrategy
        RULE 1 : IF gameState IS NOT stopped AND
ballDistance IS controllable AND (attackRisk IS NOT
high OR counterAttackRisk IS high) THEN
strategyRecommendation IS attack;

        RULE 2 : IF gameState IS stopped OR
(ballToOpponentDistance IS outOfControl AND
ballDistance IS outOfControl) THEN
strategyRecommendation IS position;

        RULE 3 : IF gameState IS NOT stopped AND
ballToOpponentDistance IS controllable AND
(counterAttackRisk IS NOT high OR attackRisk IS high)
THEN strategyRecommendation IS defend;

    END_RULEBLOCK

END_FUNCTION_BLOCK
```

A segunda etapa define a recomendação de estratégia. Ela recebe como entrada os riscos calculados na primeira etapa e mais, a menor distância entre um colega e a bola e a menor distância entre um oponente e a bola. Por fim, o estado do jogo também é utilizado, podendo ser jogo parado ou correndo. A recomendação de estratégia é produzida como resultado final da inferência.

6.2.1.2 Estabilidade da Estratégia

Uma estratégia nova implica no cálculo de novos grupos, atribuição de jogadas aos grupos e por fim execução destas jogadas por atividades e ações. Ao substituir uma estratégia por outra, é necessário que toda esta cadeia de dependências seja recalculada, abortando as jogadas atuais e escolhendo novas jogadas que se

adéqüem à nova estratégia. Desta forma, uma mudança de estratégia é um processo caro em termos de processamento. Além deste custo, mudanças freqüentes na estratégia acabam não permitindo que as jogadas em andamento cheguem ao seu final; por este motivo, mudanças de estratégia devem ser evitadas ao máximo.

O sistema de recomendação de estratégia trabalha com três valores possíveis (ataque, defesa e posicionamento) e provê uma recomendação com base em valores contínuos, de forma que existe um valor de limiar entre uma recomendação e a outra. Se a situação das variáveis de entrada permanecer muito próxima deste limiar, a estratégia recomendada irá se alterar sempre que os valores de entrada oscilarem para cima ou para baixo mesmo que de forma mínima. Esta oscilação é esperada, já que as jogadas dificilmente progridem linearmente em função das ações ou reações do time oponente. Sendo assim, o sistema de recomendação, sozinho, não proporciona a estabilidade necessária para que uma estratégia persista ao ponto de dar resultados.

Já que o objetivo de se manter uma estratégia é, basicamente, não interromper jogadas promissoras, a técnica utilizada para resolver este problema consiste em utilizar um sistema auxiliar para definir quando uma nova estratégia é necessária. Este sistema avalia o progresso das jogadas em execução por cada grupo e só invalida uma estratégia caso as jogadas sejam concluídas, seja com sucesso ou falha.

Ao utilizar dois sistemas distintos, um para iniciar uma estratégia e outro para finalizá-la, a estabilidade é assegurada.

6.2.2 Divisão em Grupos

De forma semelhante à definição da estratégia, há apenas uma tarefa responsável por dividir o time em grupos de jogadores. Este algoritmo é responsável por fazer todos os procedimentos necessários para a criação dos grupos e disparo das jogadas adequadas aos grupos encontrados.

6.2.2.1 Formação de Grupos

Os grupos são formados por proximidade entre os jogadores, bem como, pela proximidade destes aos pontos estratégicos do campo: centro, cantos e goleiras.

A regra para a divisão de grupos é:

1. Calcular as distâncias absolutas entre um jogador e todos seus companheiros de time;
2. Obter a média das distâncias;
3. Estabelecer que todos os jogadores localizados a uma distância menor do que a média fazem parte do mesmo grupo;

Verificando empiricamente que o histograma das distâncias entre os robôs obedece a uma distribuição normal, é esperado que a densidade da probabilidade de um robô se encontrar em condições similares a qualquer outro seja de 50. Desta forma, os grupos geralmente dividirão o time ao meio tendo dois grupos com 2 ou 3 jogadores cada.

6.2.2.2 Estabilidade dos grupos

Grupos são formados pelo posicionamento físico dos robôs no campo. Ao estabelecer uma relação com o ambiente físico do jogo se garante a estabilidade dos resultados, já que a posição de um jogador não irá mudar de maneira indeterminável.

Outro fator importante é que, da mesma forma como definido para a estratégia, a regra para formação de um grupo não é a mesma utilizada para desfazê-lo. Desta forma se obtém ainda mais estabilidade para a execução de uma jogada. Isto é necessário já que um grupo é formado para executar uma jogada, porém ele não pode ser desfeito antes que a jogada seja concluída ou abandonada. Ainda, por esta razão, os grupos precisam interpretar o andamento das jogadas e estar vinculados a elas.

6.2.2.3 Características dos Jogadores em um Grupo

O sistema de aglomeração varre todos os jogadores procurando por características neles através da aplicação de regras relacionadas ao posicionamento destes. As características têm dois objetivos principais. Em primeiro lugar, encontrar jogadas adequadas à situação do jogo e em segundo lugar, viabilizar ao nível de jogadas uma decisão simples quanto ao que cada jogador deve fazer dentro da jogada que participa.

A cada jogador podem ser atribuídas tantas características quantas forem detectadas nele. Segue a descrição delas por categoria.

As características de um jogador quanto à sua posição em relação ao gol adversário são:

- ADVANCED: um jogador é considerado avançado quando seu centro está à frente do meio de campo;
- WITHDRAWED: um jogador é considerado recuado quando seu centro está atrás do meio de campo;
- CROSS_KICK_RECEIVER: um jogador é considerado recebedor caso esteja dentro da área adversária;

A característica de um jogador quanto à sua posição em relação aos cantos do campo é:

- CROSSER: um jogador é considerado o cruzador caso esteja mais próximo a um dos cantos oponentes;

A característica de um jogador quanto à sua posição em relação à bola é:

- BALL: um jogador é considerado possuidor da bola se estiver mais próximo da bola do que qualquer outro robô no campo;

As características secundárias de um jogador são calculadas em função daquelas já existentes:

- LEADER: um jogador é considerado o líder de um grupo se ele for BALL ou:

  - Para estratégia de ataque ou posicionamento ele é o jogador mais avançado do grupo;
  - Para estratégia de defesa ele é o jogador mais recuado do grupo;
- GOALKEEPER: um jogador é considerado o goleiro se ele é o jogador mais recuado do grupo mais recuado (próximo ao próprio gol);

6.2.2.4 Seleção das Jogadas

Após determinar os grupos, a seleção das jogadas adequadas é feita a partir de algumas listas de jogadas. Existe uma lista de jogadas para cada estratégia, e estas listas são ordenadas pelo seu coeficiente de sucesso de forma que, caso haja mais de uma jogada candidata, aquela com maior índice de sucesso é encontrada antes e será a escolhida. Para cada grupo é feita uma busca por estas listas de acordo com a estratégia corrente e as características dos seus jogadores.

Para simplificar o entendimento será apresentado um exemplo.

Sejam definidas as seguintes Jogadas com seus respectivos requisitos:

- Ataque em Trio:

$$TripleAttack[\aleph(G) > 2, \{ADVANCED, BALL\}, Attack]$$

- Suporte a Ataque:

$$AttackSupport[\aleph(G) < 5, \{WITHDRAWED\}, Attack]$$

Estas jogadas ficam disponíveis na lista de jogadas de ataque por ordem de importância da seguinte forma:

$$Plays = \{TripleAttack, AttackSupport\}$$

Para este exemplo, a figura 6.5 apresenta à esquerda a situação do jogo e, à direita, os grupos resultantes da execução do algoritmo de divisão de grupos, com as características detectadas para cada jogador.

Figura 6.5: Exemplo da divisão do time em grupos.

Utilizando a notação definida anteriormente os grupos ficam assim:

$$G_1 = \{Gremio3_{\{WITHDRAWED\}}, Gremio2_{\{ADVANCED,BALL,LEADER\}}, Gremio4_{\{ADVANCED\}}\}$$

$$G_2 = \{Gremio5_{\{WITHDRAWED,GOALKEEPER\}}, Gremio1_{\{WITHDRAWED,LEADER\}}\}$$

Um a um, os grupos procuram por jogadas adequadas a eles. Para o grupo *G1* os requisitos para a jogada *Ataque em Trio* são satisfeitos, ou seja, $\aleph(G_1) = 3$ e as características necessárias estão presentes, logo esta jogada é selecionada.

O grupo *G2* não possui nenhum jogador com a característica ADVANCED ou BALL, logo não se enquadra com o Ataque em Trio. A próxima jogada da lista é Suporte a Ataque. Para esta, todos os requisitos são satisfeitos e portanto ela é selecionada.

6.2.3 Jogadas

Neste nível surge uma série de algoritmos de execução distintos. Em primeiro lugar, porque é preciso que o sistema conheça diferentes jogadas para diferentes situações para que possa ter autonomia. Outro motivo é decorrente da divisão do time em grupos distintos, que gera a primeira oportunidade para explorar paralelismo espacial. Cada grupo deve executar uma jogada distinta, simultaneamente.

Não há um limite para o número de jogadas possíveis, e é esperado que o amadurecimento do sistema gere a necessidade de novas jogadas. A seguir são descritas as jogadas existentes.

6.2.3.1 Ataque Individual

Configuração de requisitos:

$$IndividualAttack[\aleph(2), \{BALL, ADVANCED\}, ATTACK]$$

Para ataque individual, a coordenação coletiva consiste em fazer com que um robô conduza a bola em direção ao gol e a chute ao identificar uma oportunidade. Ao mesmo tempo um segundo robô do grupo atua como suporte, em uma posição recuada, para receber um passe caso o progresso seja ruim.

6.2.3.2 Cruzamento

Configuração de requisitos:

$$Crossing[\aleph(\geq 2), \{BALL, ADVANCED, CROSS_{KICK_{RECEIVER}}, CROSSER\}, ATTACK]$$

Para esta jogada, dois atacantes se deslocam, aqueles sem a bola vão em direção à trave oposta ao cruzador. Aquele que possui a bola a conduz em direção ao canto mais próximo da quadra e a chuta perpendicularmente de encontro ao centro da área do oponente.

6.2.3.3 *Ataque em Trio*

Configuração de requisitos:

$$TripleAttack[\aleph(>2), \{BALL, ADVANCED\}, ATTACK]$$

O ataque em trio requer um alto grau de coordenação coletiva. Para esta jogada, três jogadores trocam passes com objetivo final de posicionar dois deles em uma situação favorável a um cruzamento.

6.2.3.4 *Suporte a Ataque*

Configuração de requisitos:

$$AttackSupport[\aleph(\leq 4), \{WITHDRAWED, \neg BALL\}, ATTACK]$$

Esta jogada consiste em posicionar os jogadores do grupo de forma a bloquear o movimento dos oponentes, quando possível, ou simplesmente se posicionar aguardando um contra ataque.

6.2.3.5 *Posicionamento Para Cobrar Pênalti*

Configuração de requisitos:

$$PenaltyKickPosition[\aleph(5), \emptyset, POSITION]$$

Todos os jogadores do time se posicionam, sendo que um cobrador é escolhido, posicionando-se atrás da bola.

6.2.3.6 *Posicionamento Para Iniciar Jogo*

Configuração de requisitos:

$$InitialPosition[\aleph(5), \emptyset, POSITION]$$

Todos os jogadores do time se posicionam, um deles é escolhido e se posiciona atrás da bola.

6.2.3.7 *Defesa Individual*

Configuração de requisitos:

$$IndividualDefense[\aleph(1), \{\neg BALL\}, DEFENSE]$$

Não há coordenação coletiva nesta jogada. O jogador simplesmente vai em direção ao oponente mais próximo com o intuito de obstruir seu caminho.

6.2.4 Atividades

Atividades, de forma semelhante às jogadas requer a existência de mais de um algoritmo de execução. A seguir, as atividades são descritas de forma sucinta.

- Movimentar: consiste em se deslocar evitando obstáculos em direção a um alvo;

- Driblar: consiste em conduzir a bola evitando obstáculos em direção a um alvo e chutar quando o alvo estiver desimpedido;
- Receber passe: esta atividade coloca um robô em posição de receber um passe. Assim que a bola estiver próxima o suficiente, o driblador é acionado para mantê-la sob controle;
- Cruzar bola: cruzar a bola requer que o robô conduza a bola em direção ao canto da quadra, gire em direção à área do oponente e chute a bola;
- Bloquear trajetória: para bloquear uma trajetória basta se deslocar a dois destinos ao mesmo tempo. O vetor resultante será o ponto médio entre um e outro;
- Passar bola: posiciona o robô atrás da bola, e a chuta em direção a um alvo;

6.2.5 Ações

A seguir são apresentadas as ações pré-definidas.

6.2.5.1 Evitar Obstáculos

Evitar obstáculos é uma das ações mais importantes e está envolvida em praticamente todas as *atividades*. Ela produz como resultado um vetor que indica a direção, sentido e velocidade com que o robô deve se deslocar para evitar uma colisão com um obstáculo.

Existe uma série de técnicas que permite o planejamento de trajetórias evitando obstáculos como diagramas de Voronoi, decomposição em células e campos potenciais (SIEGWART e NOURBAKHSH, 2004). Para o sistema de controle baseado em TCs, a técnica utilizada não deve definir o melhor caminho sem obstáculos entre um ponto e outro, pois os obstáculos são móveis e o cálculo de uma trajetória longa perde sua validade muito rapidamente exigindo novos cálculos, não sendo esperado que a TC responsável pela definição do vetor de movimento do robô consuma um tempo exagerado.

O mecanismo escolhido para fazer com que um robô evite obstáculos é a técnica de campos potenciais, na qual é utilizada uma abstração de campos de força, similar aos existentes na Física para representar cargas elétricas que se atraem ou repelem. Nesta abstração, os obstáculos são representados por cargas que estabelecem, para cada ponto dentro do seu campo de ação, um potencial de repulsão inversamente proporcional ao quadrado da distância entre este ponto e o centro da carga. Este potencial gera um vetor que indica a direção, sentido e intensidade com que o robô deve se deslocar para permanecer o mais distante possível dos seus obstáculos.

Esta técnica permite que se calcule a cada instante o vetor que afasta o robô de seus obstáculos sem desperdiçar tempo estabelecendo, previamente, uma trajetória de longo prazo. O vetor resultante pode ser utilizado em combinação com outros vetores de movimento para definir um comportamento complexo, como, por exemplo, deslocar-se em direção a um alvo evitando obstáculos e conduzindo a bola.

A figura 6.6 exemplifica uma situação em que um jogador está realizando a atividade desviar obstáculos. Os obstáculos em questão são os seus companheiros de time, do lado esquerdo da quadra. Os campos potenciais fazem com que o robô

se mova em oposição aos obstáculos como pode ser visto pela seta que indica o movimento a ser realizado para que o robô se afaste de seus obstáculos. À esquerda a situação sendo tratada com os obstáculos circulados. A seta indica a direção do movimento em resposta aos campos potenciais repulsivos resultantes, sendo representados à direita.

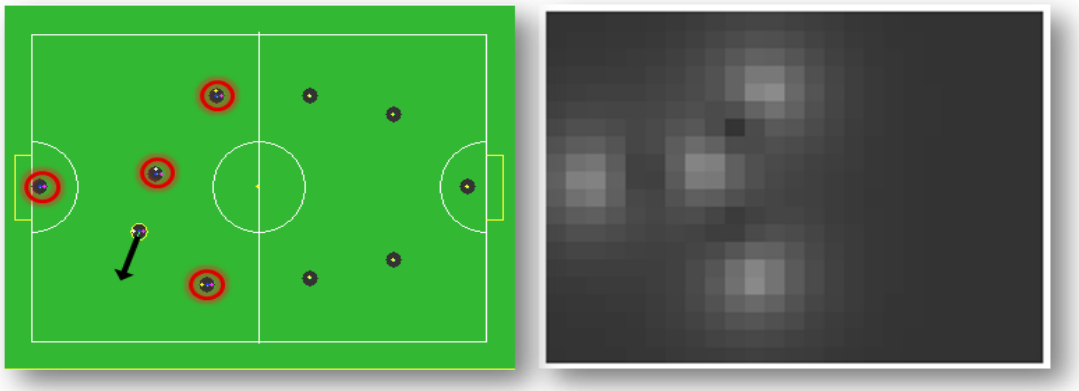


Figura 6.6: Campos potenciais calculados para uma situação de um jogo.

6.2.5.2 Andar em Direção a Um Alvo

De forma semelhante à ação que evita obstáculos, fazer com que um robô ande em direção a um alvo é feita através de campos potenciais. Para esta ação, o alvo é representado por uma carga e o campo, neste caso, atrai o robô em sua direção. A figura 6.7 demonstra a ação de ir em direção à bola e o efeito que a carga que representa a bola exerce em um jogador. À esquerda, a figura mostra um robô indo em direção à bola em função de um campo de atração. Em cor clara está a área onde a intensidade do campo é maior. Em preto a localização do alvo.

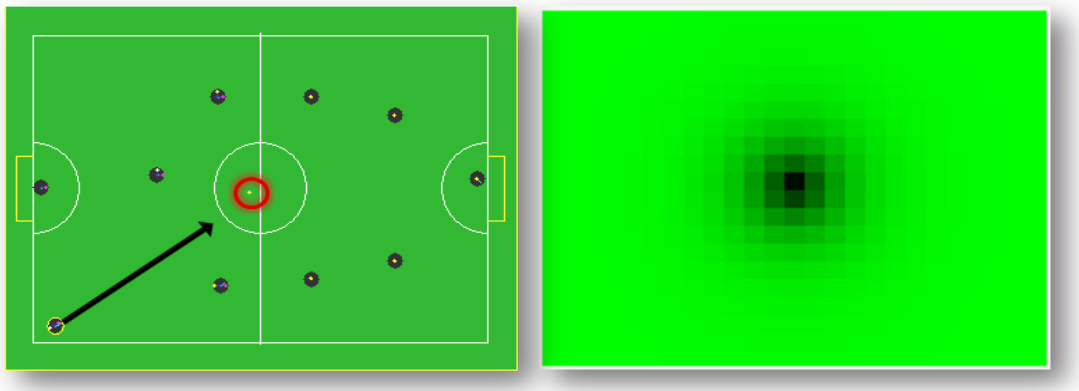


Figura 6.7: Campo potencial que atrai o robô em direção à bola.

6.2.5.3 Carregar a Bola

Carregar a bola é uma ação complexa para ser modelada, já que requer ajuste constante e preciso do robô para que não perca o controle sobre a trajetória da bola.

Para realizar a ação de carregar a bola são utilizados três campos potenciais, um destes campos faz com que o robô seja atraído para trás da bola, os outros estabilizam a trajetória do robô repelindo-o da frente da bola. À medida que o robô se posiciona atrás da bola, o campo potencial se desloca ligeiramente à frente da bola, fazendo com que o robô avance e a empurre. A combinação destes três campos produz um movimento que está constantemente avançando ajustando com exatidão a posição do robô, como é mostrado na figura 6.8. À esquerda um jogador está com a bola em seu controle e a conduz em direção a um alvo. À direita os campos potenciais utilizados nesta ação.

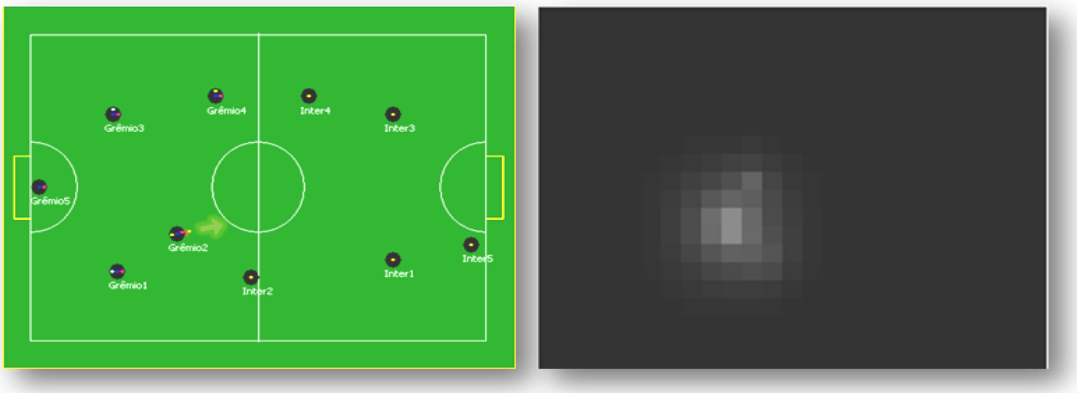


Figura 6.8: Campos potenciais que permitem ao jogador manter o controle da bola.

6.2.5.4 *Chutar Bola*

Chutar bola é uma ação que alinha o robô com a bola e seu alvo e dispara o chutador do robô quando a distância e o ângulo do jogador em relação à linha estabelecida pela bola e seu alvo forem adequados.

6.2.5.5 *Receber Passe*

Esta ação ajusta a direção do robô apontando-o para a bola, definindo a direção e velocidade com que o robô deve se deslocar para interceptar a trajetória da bola através da projeção de sua posição futura. A figura 6.9 é mostrada a trajetória da bola e projeção perpendicular a partir da posição do robô, definindo a rota a ser percorrida para interceptar a bola.

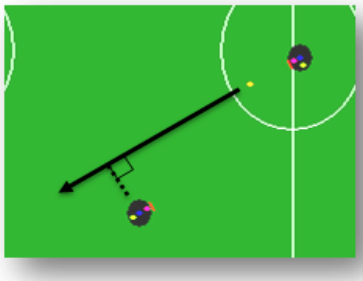


Figura 6.9: Receber passe.

6.2.6 Movimento

O nível de movimento é composto por uma única tarefa de um tipo especial. Esta tarefa envia os resultados de dentro do sistema de TCs para o mundo exterior. Por este motivo, há uma mistura de lógica seqüencial e paralela nesta tarefa. Ao mesmo tempo em que ela recebe estímulos do nível de atividades de forma assíncrona, ela também possui comportamento iterativo enviando comandos para os robôs a uma freqüência de 20 Hz, obtida por meio de uma tarefa auxiliar que atua como relógio. Os estímulos ativam a tarefa que, por sua vez, envia os comandos pendentes aos robôs. Desta forma, o tempo de resposta dos robôs é curto o suficiente para manutenção das jogadas em tempo hábil e, ao mesmo tempo, é criada uma janela de 50 ms para que resultados sejam recebidos das ações e consolidados antes de serem enviados aos robôs.

O valor de 20 Hz (50 ms) foi utilizado em função de ser o tempo de resposta normalmente utilizado pelo sistema de rádio dos robôs, porém este valor é configurável e pode ser ajustado para diferentes plataformas se necessário.

O protocolo de transmissão de comandos aos robôs estabelece que cada comando enviado, atua sobre um robô por vez, passando um conjunto de configurações a este. Os comandos são enviados utilizando um laço simples que itera por todas as requisições pendentes para cada robô e as envia uma a uma.

Um comando possui as seguintes configurações:

- Velocidade de deslocamento: um vetor bidimensional que determina a direção e intensidade da velocidade desejada de um robô em m/s;
- Velocidade de rotação: um valor real que indica a velocidade de rotação do robô sobre seu eixo radial em rad/s;
- Habilitar/desabilitar driblador: um valor booleano que indica se o driblador está habilitado ou não;
- Habilitar chutador: um valor real que indica a força com que o chutador deve ser disparado;

Apenas as configurações desejadas precisam ter um valor atribuído, aquelas que receberem um valor nulo permanecerão inalteradas.

6.2.7 Leitura de Estado

Para a leitura de estado uma única tarefa que monitora o estado do jogo atualizando a sua estrutura lógica a cada 30 ms (33,3 Hz). Sua natureza é similar à da TC de movimento, realizando interface com o mundo externo por meio de comportamento serial iterativo e estimulando as tarefas subseqüentes de forma assíncrona.

O valor de 33,3 Hz (30 ms) foi utilizado em função de ser o tempo de resposta normalmente utilizado pelo sistema de visão dos robôs, porém este valor é configurável e pode ser ajustado para diferentes plataformas se necessário.

Uma série de dados sobre cada jogador e sobre a bola é lida e armazenada formando um modelo lógico da situação do jogo. Todos os elementos móveis do

jogo (bola e robôs) compartilham uma estrutura de dados similar. Para estes, os seguintes dados são monitorados:

- Nome;
- Posição;
- Ângulo;
- Velocidade;
- Velocidade de rotação;

Há ainda um conjunto de dados específicos aos robôs sendo composto por:

- Time;
- Estado do driblador;
- Estado do chutador;

6.3 Estrutura de Tarefas Conexionistas

Cada um dos algoritmos de execução descritos anteriormente é implementado dentro de classes em linguagem Java que estendem a classe fundamental das Tarefas Conexionistas. Em tempo de execução, estas tarefas precisam ser instanciadas de acordo com a situação do jogo. É normal que várias instancias de uma mesma ação estejam em execução simultaneamente para atingir objetivos de diferentes atividades e jogadas; porém é inviável manter todas as combinações válidas de jogadas, atividades e ações, constantemente em execução, e utilizar apenas aquelas necessárias em um determinado momento. Fazer isto acarretaria um enorme esforço do agendador de tarefas sem trazer nenhum benefício em troca, já que, boa parte das tarefas em execução não estaria em atividade.

Para estabelecer o que deve estar em execução em cada momento, é preciso que as tarefas de nível mais alto possuam em sua definição, quais as tarefas de que dependem para atingir seus objetivos. Uma vez definidas as tarefas, é também sua responsabilidade instanciá-las e colocá-las em execução. Esta divisão de responsabilidades gera uma estrutura aninhada de criação de tarefas, pois cada uma define e gerencia suas dependentes imediatas.

O mecanismo descrito anteriormente permite que apenas as tarefas necessárias fiquem em execução. No sistema de controle autônomo para futebol de robôs, apenas quatro tarefas estão permanentemente em execução. São elas: Estratégia, Divisão de Grupos, Movimento e Leitura de Dados. Todas as demais são executadas apenas quando solicitadas.

O diagrama da figura 6.10 apresenta a maneira como a execução de uma jogada é gerenciada pelo sistema utilizando a jogada *Cruzamento* como exemplo. A jogada instancia duas atividades, *Receber Cruzamento* e *Cruzar Bola*. Estas atividades, por sua vez, instanciam suas respectivas ações.

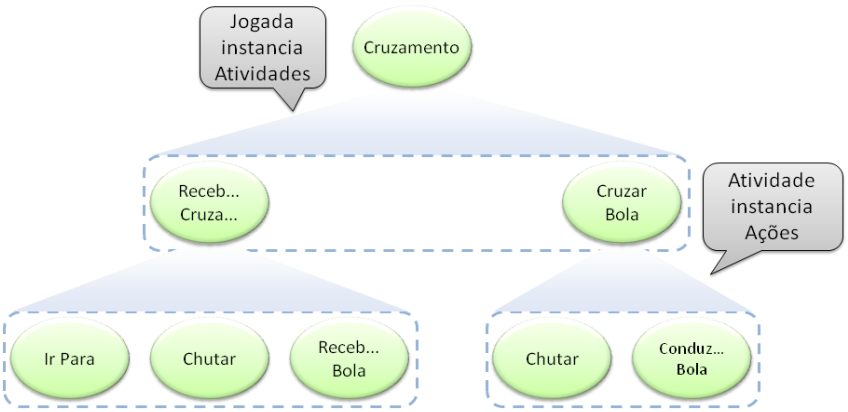


Figura 6.10: Diagrama representando a criação de tarefas e sub-tarefas.

Além disto, é necessário que as TCs destruam as tarefas que não são mais necessárias. Isto é feito de forma similar à sua criação, ou seja, quando for identificada a necessidade de que uma nova jogada substitua uma jogada anterior, a jogada anterior é destruída e suas dependências também o são.

Para que uma TC acompanhe o andamento das tarefas situadas em níveis inferiores a ela, é preciso que as monitore. Isto é feito conectando as TCs para estabelecer um canal de comunicação que permite às tarefas reportarem as mudanças em seus estados para as TCs situadas em níveis superiores. Uma TC superior deve controlar as inferiores para que corrijam sua atuação, logo é necessário também, um segundo canal de comunicação, que vai das superiores para as inferiores e conduz os comandos de ajuste.

Usando a mesma jogada do exemplo anterior, a figura 6.11 demonstra as conexões entre TCs através de um grafo direcionado que indica o sentido dos eventos. As setas contínuas representam os eventos de mudança no estado de uma TC, geralmente fluindo de um nível inferior para um superior. As setas tracejadas são eventos de controle que alteram o comportamento do receptor, geralmente fluindo de um nível superior para um inferior.

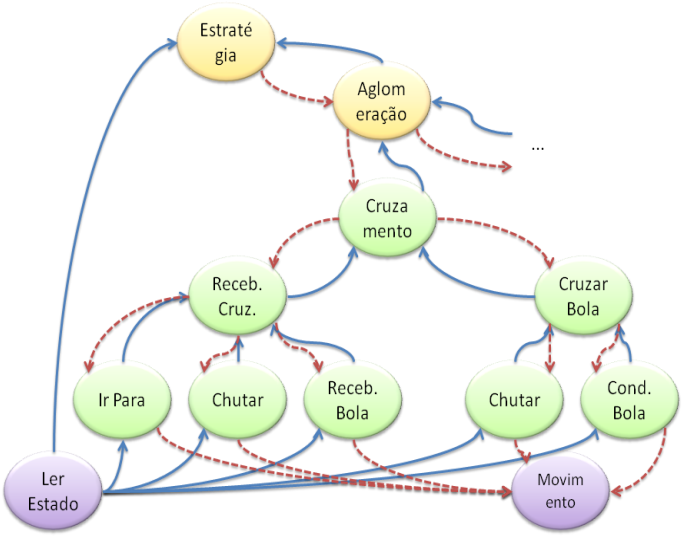


Figura 6.11: Conexões entre TCs para realizar uma jogada de cruzamento.

6.4 Trabalhos Relacionados

A seguir são apresentados alguns trabalhos relacionados a este, dentro da área de futebol de robôs.

Em *A Multi-threaded Approach to Simulated Soccer Agents for the RoboCup Competition* (KOSTIADIS e HU, 2000), é apresentado um sistema multi-agentes baseado em múltiplas threads para aplicação na liga simulada da Robocup. Cada agente consiste de uma aplicação com três threads. Uma *thread* é responsável por pensar (*Think*), outra por agir (*Act*) e uma terceira por sentir (*Sense*). As threads se comunicam através de um modelo de mundo situado em um buffer de memória compartilhada O acesso à memória é sincronizado por monitores. A *thread* Sense responde mudanças no jogo e atualiza os dados do modelo lógico do mundo. A *thread Act* é ativada a cada 100 ms para enviar comandos pendentes ao seu robô. A *thread Think* está em execução permanente e analisa os dados obtidos por *Sense* para produzir comandos ao robô.

No trabalho *The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team*, (BOER e KOK, 2002) desenvolvido como dissertação de mestrado no curso de Ciência da Computação da Universidade de Amsterdam, são apresentados os passos envolvidos no desenvolvimento de um time de futebol de robôs para a liga simulada da Robocup. Seu objetivo é servir como um guia prático para equipes iniciantes, e de forma semelhante ao desenvolvido neste trabalho, são apresentadas todas as técnicas utilizadas para a criação de um sistema de controle. O seu foco é diferente do proposto neste trabalho, pois utilizar a abordagem multi-agentes, porém grande parte dos mecanismos apresentados possui semelhança com os utilizados neste trabalho. Como exemplo, é possível citar a divisão do controle dos robôs em níveis compatíveis com a estrutura básica da célula robótica inteligente (JACAK, 1999) e ainda, a divisão de tarefas para execução paralela por meio de threads.

De um ponto de vista de paralelismo, esta proposta é similar à apresentada em (KOSTIADIS e HU, 2000). Dentro do sistema proposto, cada agente é uma aplicação com três threads. Uma *thread* é responsável por pensar (*Think*), outra por agir (*Act*) e uma terceira por sentir (*Sense*). As *threads* se comunicam através de um modelo de mundo situado em um buffer de memória compartilhada O acesso à memória é sincronizado por monitores.

7 TESTES E RESULTADOS

Este capítulo apresenta uma visão geral das métricas de desempenho para algoritmos paralelos. A seguir são apresentados os testes do modelo de programação e por fim os testes do sistema de controle autônomo desenvolvidos neste trabalho.

7.1 Métricas de Desempenho

Existe uma série de formas para avaliar o desempenho de aplicações no que diz respeito ao paralelismo. As métricas mais utilizadas tem como base a lei de Amdhal que procura comparar o desempenho de uma aplicação serial com uma paralela. Amdhal diz que "O ganho de velocidade de um programa usando múltiplos processadores em computação paralela é limitado pelo tempo gasto pela fração seqüencial do programa.", ou seja, se metade do programa precisa ser executada de forma seqüencial, o maior ganho de velocidade (*speedup*) que pode ser obtido pela sua execução em múltiplos processadores é de 50%.

A lei de Amdhal é representada na seguinte fórmula, onde se considera que o numero de processadores disponíveis é ilimitado ($p \to \infty$):

$$speedup = \frac{1}{1-f}$$

onde f corresponde à fração paralelizável do programa.

A figura 7.1 apresenta a evolução do *speedup* em relação à porção paralelizável de código de um programa.

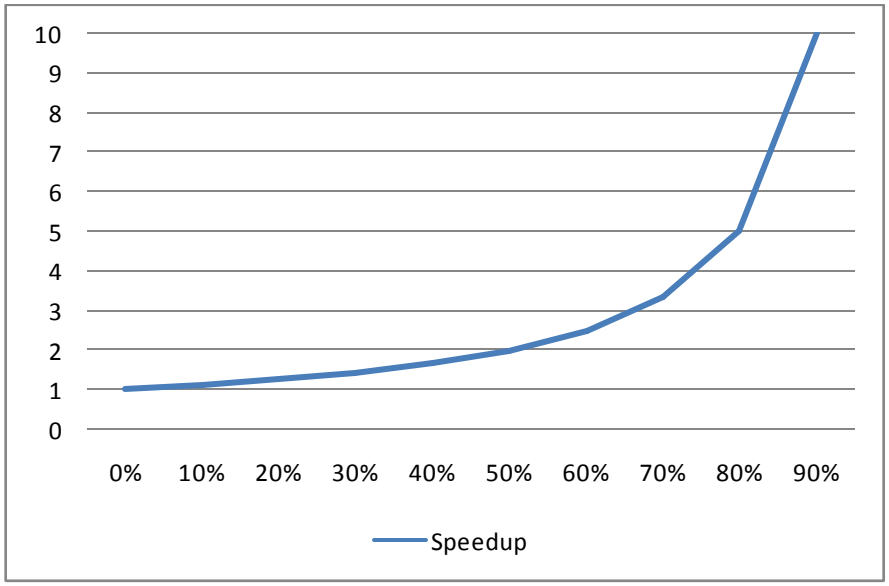


Figura 7.1: *Speedup* em função da porção paralelizável do código.

Introduzindo um número limitado de processadores desenvolvendo a fração paralelizável, se obtém a seguinte equação (BARNEY, 2010):

$$speedup = \frac{1}{\frac{f}{p} + s}$$

Onde *f* corresponde à fração paralelizável do programa, *s* é a fração seqüencial (*1 - f*) e *p* é o número de processadores.

Esta equação produz o gráfico mostrado na figura 7.2.

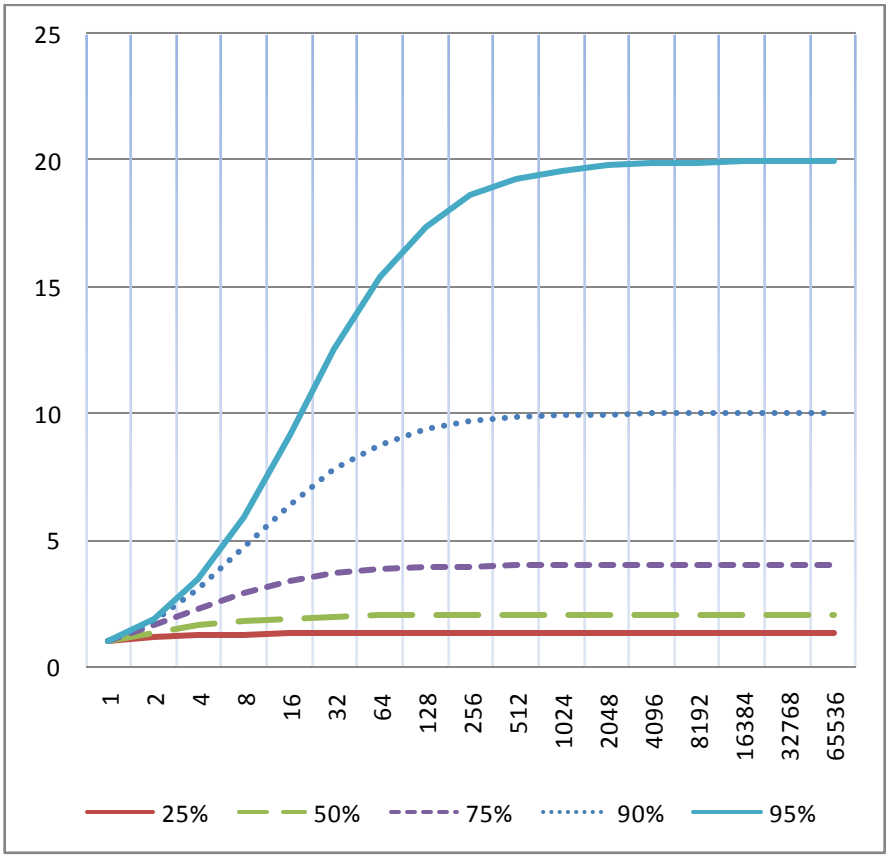


Figura 7.2: Relação entre *speedup* e número de processadores em função da porção paralelizável do código.

Quando enunciada, em 1967, esta lei servia para comprovar que o ganho de desempenho não pode ser obtido arbitrariamente pelo emprego de mais recursos paralelos, porém ela permite também quantificar o paralelismo de uma maneira bastante clara, basta para isto fazer alguns ajustes na fórmula como será descrito a seguir (RAUBER e RÜNGER, 2010).

Juntamente com o s*peedup*, outra medida bastante utilizada para medir o paralelismo se chama Eficiência.

Speedup, portanto, representa o aumento de velocidade quando se executa um processo em p processadores em relação à velocidade de execução deste processo em apenas um processador.

Como não é simples definir, de forma clara, a fração seqüencial e a fração paralelizável de um programa, um modo mais útil de medir o *speedup* utiliza a lei do trabalho (LEISERSON e MYRMAN, 2008) que define o *speedup* em função do tempo de processamento.

Neste caso, o cálculo do s*peedup* é feito utilizando a fórmula

$$speedup(p) = \frac{T(1)}{T(p)}$$

Onde $T(n)$ é o tempo de execução em um número n de processadores e p é o número de processadores sendo testados.

Um programa que apresenta *speedup* linear é aquele em que o *speedup* cresce linearmente em relação ao número de processadores.

Eficiência (EAGER, ZAHORJAN e LAZOWSKA, 1989) é a relação entre o *speedup* e o número de processadores, e é calculada por:

$$eficiência = \frac{speedup(p)}{p}$$

Em um programa ideal, onde o *speedup* é igual ao número de processadores, a eficiência do paralelismo teria valor 1 (equivalendo a 100%).

Além destas duas métricas, um dado relevante para a o modelo de programação proposto, é o custo do gerenciamento dos eventos pela plataforma de concorrência. Este custo define a o custo mínimo do algoritmo de processamento de eventos de forma que a comunicação não tome mais tempo do que a execução de código funcional.

Os testes apresentados a seguir se dividem em testes do modelo de programação e testes do sistema de controle.

7.2 Testes do Modelo de Programação

Os testes do modelo de programação foram conduzidos de maneira a avaliar as medidas de *speedup* e desempenho que podem ser obtidos por meio do uso do modelo de programação proposto, e, alem disto, verificar como o balanceamento de carga e a comunicação afetam o paralelismo em um programa que utiliza este modelo.

Para cada teste apresentado, foi realizada uma bateria de execuções e obtida a média dos seus resultados. O número de execuções realizado está definido na descrição de cada um dos testes.

Para avaliar o desempenho do modelo de programação proposto foram utilizados dois ambientes de hardware, descritos a seguir:

Máquina 1:

- Processador Core 2 Duo P8600 com dois núcleos;
- 3 GB de memória DDR2-800;
- Sistema Operacional Windows 7 Home Basic;
- Java Runtime Environment 1.6;

Máquina 2:

- 2 Processadores Intel Xeon E5530 Quad-Core com HyperThreading (SMT);
- 12 GB de memória DDR2-800;

- Sistema Operacional Ubuntu 6.0;
- Java Runtime Environment 1.6;

A Máquina 2 foi gentilmente cedida pelo Grupo de Processamento Paralelo e Distribuído (GPPD) do Instituto de Informática da UFRGS para a realização destes testes.

A execução dos testes foi feita variando o número de CPUs disponíveis através do fornecimento de uma máscara de afinidade de processo ao sistema operacional. Esta máscara estabelece quais CPUs um processo pode utilizar, limitando seu processamento aos mesmos. Desta maneira, em uma única máquina com múltiplas CPUs é possível simular um número menor de CPUs permitindo comparações entre as diferentes execuções.

Para os testes, foi desenvolvido um programa especifico que possui características próximas às de um sistema ideal, permitindo a avaliação direta da capacidade do modelo de programação. Este programa permite, ainda, que se alterem algumas configurações como o tempo de execução de cada TC e o volume de dados carregado por um estímulo, permitindo a análise dos sintomas de um programa com problemas de modelagem.

O programa é capaz de resolver duas equações semelhantes. Cada equação possui um conjunto de operações proporcional ao número de processadores disponíveis no ambiente em que o seu teste é executado. A equação A é destinada à máquina 1 e a equação B à máquina 2:

A. $[(a+b)\times c]\times(d+e)\times(f+g)$
B. $[(a+b)\times c]\times(d+e)\times(f+g)+[(a+b)\times c]\times(d+e)\times(f+g)$

A intercalação entre somas e multiplicações torna a equação não-associativa, isto é, a ordem em que as operações são agrupadas interfere no seu resultado. Ao eliminar a associatividade, é criada uma barreira à execução paralela espacial, exigindo o uso de paralelismo temporal.

Dois algoritmos são utilizados para resolver cada uma das equações em questão. Um dos algoritmos realiza o cálculo de maneira tradicional, em série. O segundo algoritmo modela a equação em Tarefas Conexionistas e resolve o problema de maneira paralela.

Para exercitar o mecanismo de comunicação entre as Entidades Conexionistas, a equação inteira, é resolvida diversas vezes, em um mesmo experimento, forçando a troca de estímulos entre as TCs.

Por fim, para elevar o consumo de CPU de maneira a obter resultados mais simples de analisar, cada operação aritmética pode ser realizada uma série de vezes, em função do experimento sendo executado.

Como saída, a aplicação produz:

1. O tempo decorrido para executar o algoritmo seqüencial;
2. O resultado do cálculo para o algoritmo seqüencial;

3. O tempo decorrido para executar o algoritmo paralelo;
4. O resultado do cálculo para o algoritmo paralelo;

7.2.1 Algoritmo seqüencial

O algoritmo seqüencial consiste em percorrer a equação da esquerda para a direita, respeitando a prioridade estabelecida pelos parênteses. Cada operação é realizada em uma seqüência, seu resultado intermediário é armazenado e, a seguir, a próxima operação é executada.

7.2.2 Algoritmo paralelo

Para o algoritmo paralelo, a modelagem em TCs foi feita fragmentando a equação em estágios, de acordo com a ordem pré-estabelecida das operações. Operações que podem ser realizadas em paralelo em um mesmo estágio foram então agrupadas em níveis. Para a equação A, existem três níveis.

O nível L1 realiza as operações entre as variáveis a, b, d, e, f e g, produzindo variáveis intermediárias h, i e j.

O nível L2 realiza operações entre h, c, i e j para produzir as variáveis intermediárias k e l.

Por fim, o nível L3 realiza operações entre k e l para produzir a variável final m.

Para a equação B, a estrutura básica é replicada e um nível adicional é necessário. O nível L4 opera com a variável m do lado esquerdo com a variável m2 do lado direito como mostra a figura 7.3. A divisão obtida para a equação A pode ser vista nas setas tracejadas.

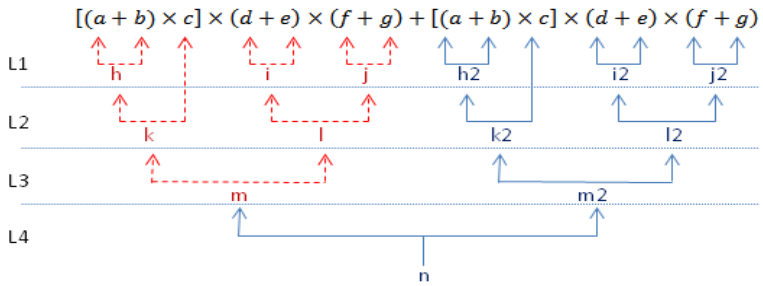


Figura 7.3: Divisão das operações da equação B em estágios e seu agrupamento em níveis.

A topologia da rede de TCs utiliza a divisão de tarefas obtida anteriormente ficando como apresentado na figura 7.4. Nesta topologia, cada entidade opera sobre duas entradas. Estas entradas são fornecidas pelo nível imediatamente inferior. Ao final, a entidade N produz notifica o sistema do término da operação fornecendo a sua saída para a tarefa E/S. Salientado, em cor escura, a estrutura para resolução da equação A. Há apenas uma tarefa que realiza comunicação com o mundo externo, responsável pela entrada e saída de dados do sistema. Esta tarefa está representada como E/S.

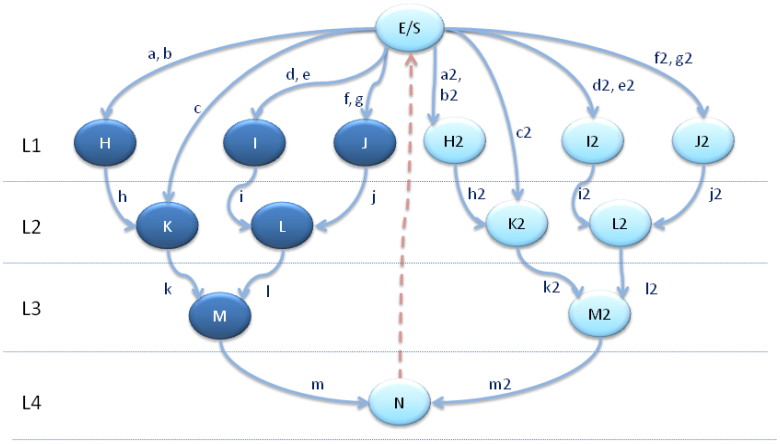


Figura 7.4: Estrutura de Tarefas Conexionistas para a equação B.

7.2.3 Teste 1: Resolução da Equação A na Máquina 1

Este teste foi executado utilizando a Máquina 1 limitando o processo à utilização de apenas 1 CPU e depois a 2 CPUs. A equação é resolvida 250 vezes e cada operação aritmética é executada 20.000.000 vezes.

7.2.3.1 Passos:

1. Definir a afinidade do processo habilitando 1 CPU;
2. Invocar o programa;
3. Coletar o resultado;
4. Definir a afinidade do processo habilitando 2 CPUs;
5. Invocar o programa;
6. Coletar o resultado;
7. Realizar os passos anteriores novamente totalizando 4 execuções;
8. Computar a média do tempo de execução;

7.2.3.2 Resultados:

Os tempos médios de execução dos algoritmos tradicional e paralelo para a Máquina 1 estão listados na tabela 7.1 abaixo.

Tabela 7.1: Tempo médio de execução de cada um dos algoritmos.

CPUs	Alg. Serial (ms)	Alg. Paralelo (ms)	Diferença (ms)	Diferença (%)
1	15420,4	15465	-44,6	-0,29%
2	15546,8	8199,2	7347,6	47,26%

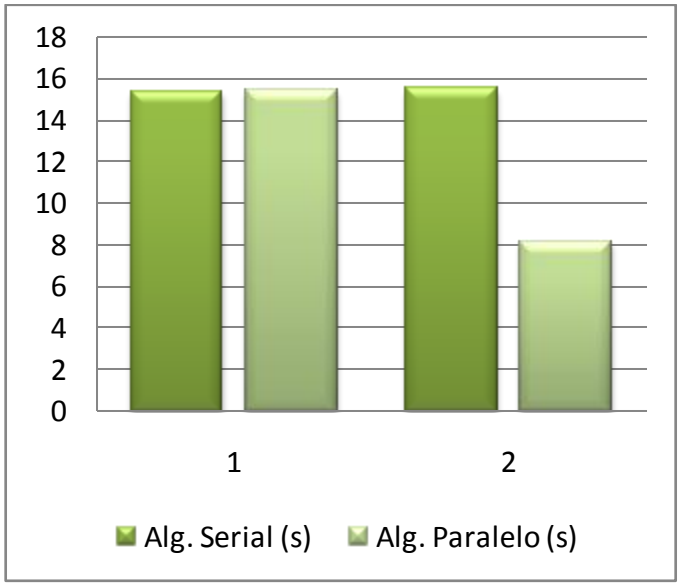


Figura 7.5: Comparativo entre os tempos de execução dos algoritmos serial e paralelo. O eixo y mostra o tempo em segundos (s) e o eixo x, o número de processadores utilizados.

Tabela 7.2: Valores para speedup e eficiência do algoritmo paralelo em uma e duas CPUs.

CPUs	Speedup	Eficiência
1	0,997	0,997
2	1,896	0,948

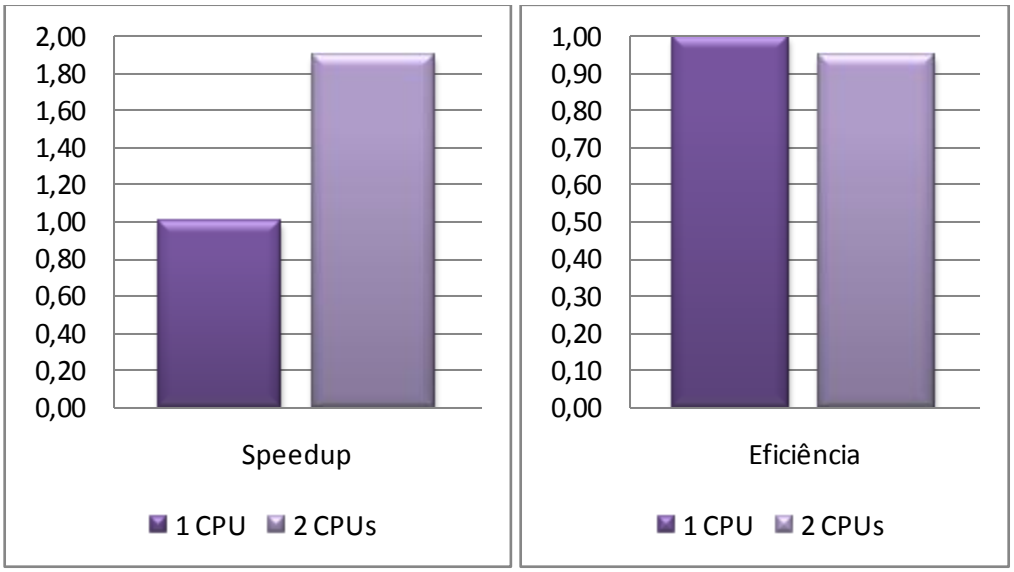


Figura 7.6: Comparativo entre o speedup e a eficiência do algoritmo paralelo em uma e duas CPUs.

7.2.3.3 Análise:

Para este teste o sistema apresentou melhoria de cerca de 47% quando comparado ao algoritmo seqüencial, ao ser executado em duas CPUs, demonstrando que o modelo permite a execução das tarefas de forma paralela na arquitetura em questão.

7.2.4 Teste 1: Resolução da equação A na Máquina 2

Este teste foi executado utilizando a Máquina 2 limitando as CPUs disponíveis, sucessivamente, de 1 CPU até 8 CPUs. A equação é resolvida 250 vezes e cada operação aritmética é executada 20.000.000 de forma a aumentar o consumo de CPU para tornar os resultados mais evidentes.

7.2.4.1 Passos

1. Definir a afinidade do processo habilitando 1 CPU;
2. Invocar o programa;
3. Coletar o resultado;
4. Realizar os passos anteriores novamente totalizando 4 execuções;
5. Aumentar o número de CPUs habilitadas para o processo em 1 e repetir todos os passos até que se atinja o número total de 8 CPUs habilitadas;
6. Computar a média do tempo de execução;

7.2.4.2 Resultados

Tabela 7.3: Tempo médio de execução de cada um dos algoritmos.

CPUs	Alg. Serial (s)	Alg. Paralelo (s)	Diferença (s)	Diferença (%)
1	22,9124	23,0064	-0,094	-0,41%
2	22,6064	11,8298	10,7766	47,67%
3	22,6272	8,1658	14,4614	63,91%
4	22,6038	6,5584	16,0454	70,99%
5	22,6164	6,018	16,5984	73,39%
6	22,594	4,511	18,083	80,03%
7	22,6498	4,29	18,3598	81,06%
8	22,5928	4,2282	18,3646	81,29%

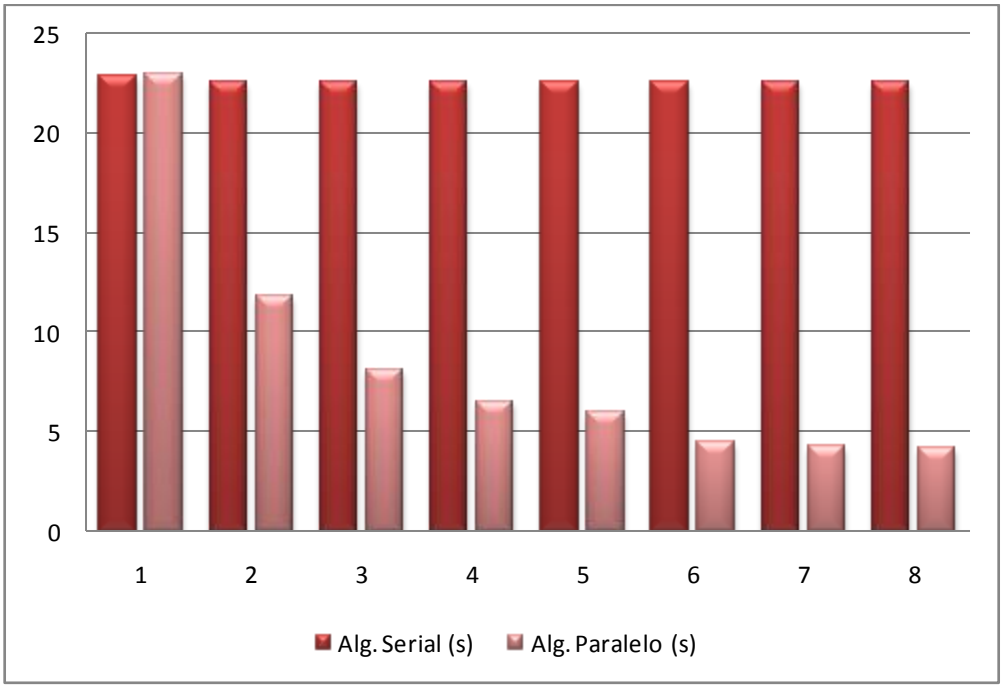


Figura 7.7: Comparativo entre os tempos de execução dos algoritmos serial e paralelo. O eixo y mostra o tempo em segundos (s) e o eixo x, o número de processadores utilizados.

Tabela 7.4: Valores para speedup e eficiência do algoritmo paralelo em uma a oito CPUs.

CPUs	Speedup	Eficiência
1	0,996	1,00
2	1,911	0,96
3	2,771	0,92
4	3,447	0,86
5	3,758	0,75
6	5,009	0,83
7	5,280	0,75
8	5,343	0,67

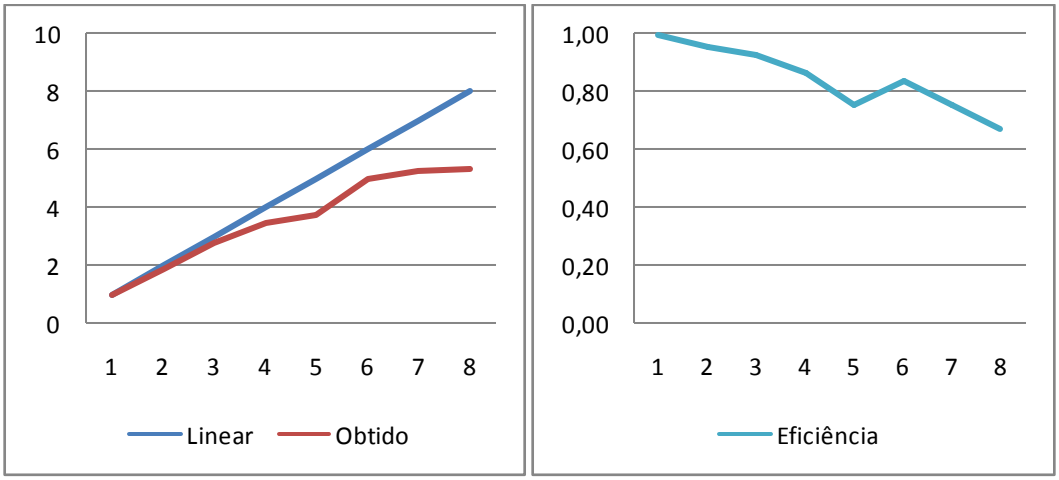


Figura 7.8: Comparativo entre o speedup e a eficiência do algoritmo paralelo em uma a oito CPUs.

7.2.4.3 Análise

De forma consistente aos resultados apresentados para a Máquina 1, para este teste o sistema apresentou uma melhoria de cerca de 47% comparado ao algoritmo seqüencial, quando executado em duas CPUs, O ganho de desempenho foi significativo até se atingirem 5 CPUs; a partir daí a evolução estagnou, sendo que ao se adicionarem mais 3 CPUs o desempenho aumentou apenas 1,2%.

A partir dos dados coletados é possível verificar, também, a existência de paralelismo temporal, já que existem 6 tarefas em execução, cada uma com um tempo de execução similar, porém com dependências entre si.

Se apenas paralelismo espacial fosse possível, cada nível teria suas tarefas executando em paralelo, mas seria necessário que cada nível aguardasse pelos resultados do nível anterior antes de poder prosseguir. Supondo que cada tarefa consome um intervalo de tempo de valor x, a execução respeitaria a seguinte proporção:

- **Nível 1:** 3 tarefas executando em paralelo, consumindo x;
- **Nível 2:** 2 tarefas executando em paralelo, consumindo x;
- **Nível 3:** 1 tarefa, consumindo x;

Tempo total de 3x, ou seja, uma melhoria de 50% comparada a uma execução em série que consumiria 6x. Para se obter resultados melhores, é preciso fazer com que todas as tarefas fiquem executando constantemente. Isto é feito simplesmente fornecendo novos dados ao nível 1 assim que ele terminar de processar os dados anteriores. Desta forma um pipeline é formado e enquanto o nível 3 finaliza a solução do primeiro conjunto de dados, o nível dois já está trabalhando no segundo conjunto e o nível um no terceiro.

O experimento demonstra que o ganho de desempenho chega a 82%, provando que há execução simultânea entre os níveis, evidenciando paralelismo temporal.

7.2.5 Teste 1: Resolução da equação B na Máquina 2

Com objetivo de validar o uso de SMT, foram acrescentadas mais 7 Threads totalizando 14, desta forma, utilizando as 8 CPUs físicas e mais as 8 CPUs lógicas. Este teste foi executado utilizando a Máquina 2 limitando as CPUs disponíveis, sucessivamente, de 1 CPU até 16 CPUs. A equação é resolvida 250 vezes e cada operação aritmética é executada 20.000.000 vezes.

7.2.5.1 Passos

1. Definir a afinidade do processo habilitando 1 CPU;
2. Invocar a aplicação exemplo;
3. Coletar o resultado;
4. Realizar os passos anteriores novamente totalizando 4 execuções;
5. Aumentar o número de CPUs habilitadas para o processo em 1 e repetir todos os passos até que se atinja o número total de 16 CPUs habilitadas (8 reais e 8 lógicos);
6. Computar a média do tempo de execução;

7.2.5.2 Resultados

Os tempos médios de execução dos algoritmos serial e paralelo para a Máquina 2 estão listados na tabela 7.5 abaixo.

Tabela 7.5: Tempo médio de execução de cada um dos algoritmos.

CPUs	Alg. Serial (s)	Alg. Paralelo (s)	Diferença (s)	Diferença (%)
1	39,5248	39,883	-0,3582	-0,91%
2	39,071	20,3154	18,7556	48,00%
3	39,1356	13,7102	25,4254	64,97%
4	39,1412	10,3564	28,7848	73,54%
5	39,1468	8,6928	30,454	77,79%
6	39,2158	7,571	31,6448	80,69%
7	39,1846	6,757	32,4276	82,76%
8	39,148	6,6308	32,5172	83,06%

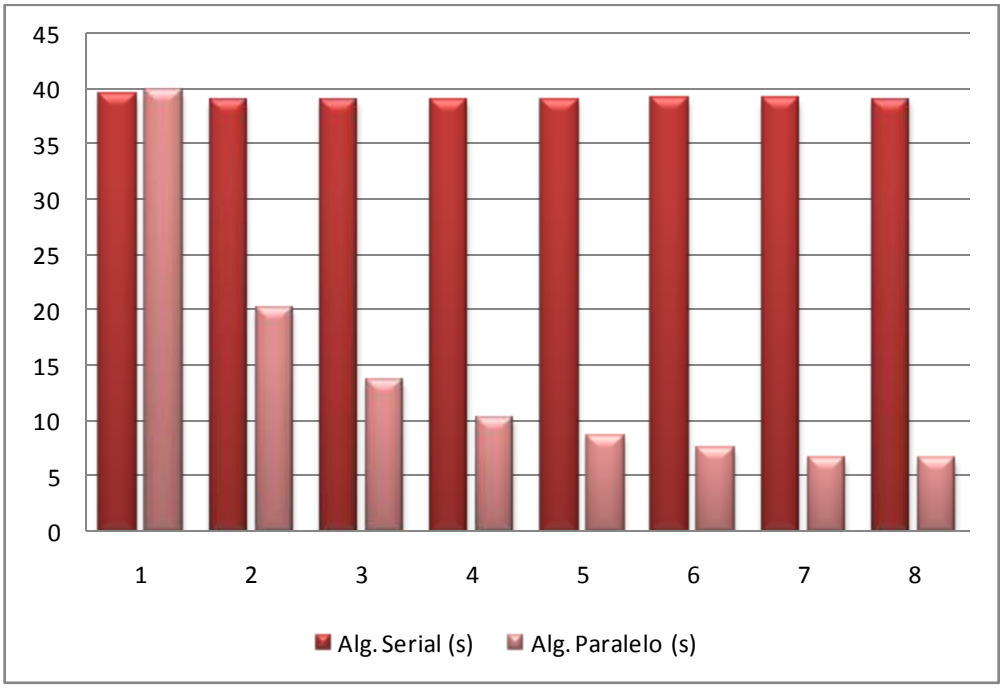


Figura 7.9: Comparativo entre os tempos de execução dos algoritmos serial e paralelo. O eixo y mostra o tempo em segundos (s) e o eixo x, o número de processadores utilizados.

Tabela 7.6: Valores para speedup e eficiência do algoritmo paralelo em uma a oito CPUs.

CPUs	Speedup	Eficiência
1	0,991	0,99
2	1,923	0,96
3	2,854	0,95
4	3,779	0,94
5	4,503	0,90
6	5,180	0,86
7	5,799	0,83
8	5,904	0,74

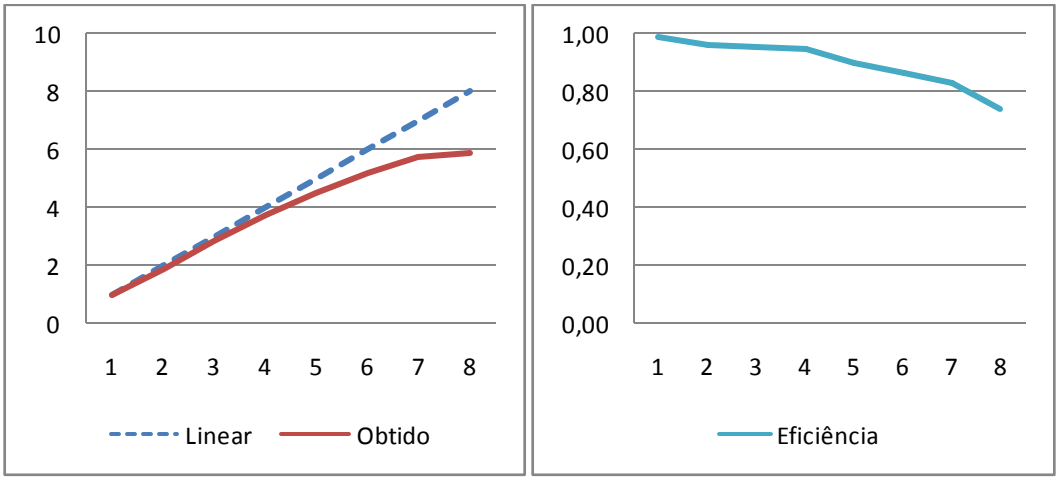


Figura 7.10: Comparativo entre o speedup e a eficiência do algoritmo paralelo em uma a oito CPUs.

A Máquina 2 possui o recurso SMT, que duplica o número de processadores lógicos. Este experimento foi realizado utilizando este recurso, e os resultados são apresentados a seguir.

Tabela 7.7: Tempo da execução,em segundos, do teste utilizando 8 CPUs com SMT e sem SMT (Normal).

Normal	SMT
6,6308	6,1024

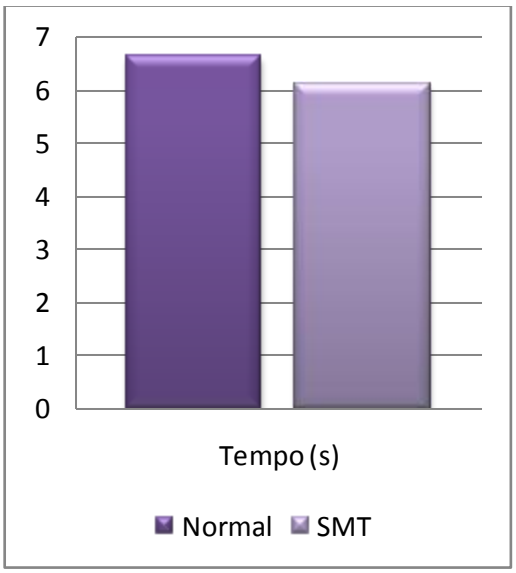


Figura 7.11: Comparação entre o tempo de execução com SMT e sem SMT (Normal).

7.2.5.3 Análise

De forma muito semelhante ao teste anterior, o desempenho fica estagnado ao se atingir o limite de 8 CPUs. Ao tentar utilizar mais de uma thread por CPU através das CPUs lógicas providas por SMT, o ganho de desempenho foi mínimo, comprovando que para os algoritmos em questão o uso de SMT não representa melhorias significativas. Este resultado era esperado, uma vez que as tarefas fazem uso intensivo de CPU com pouco acesso à memória, e possuem poucos pontos de decisão onde erros de *branch prediction* poderiam acontecer. Mesmo assim, isto não significa que a arquitetura proposta não possa tirar proveito desta tecnologia, já que o uso de SMT está mais vinculado aos algoritmos executados em cada tarefa do que à infra-estrutura de execução destas.

Os valores obtidos para speedup estão na faixa de 6 quando utilizados 8 CPUs. Com estes valores é possível calcular a fração seqüencial do código por meio das formulas apresentadas no inicio deste capítulo e concluir que existe uma fração de 94,9% do código que é paralelizável, e uma fração de 5,1% que é seqüencial.

Considerando a natureza do problema sendo resolvido, este valor permite afirmar que a infra-estrutura de suporte ao modelo de programação desenvolvido impõe ao sistema uma fração seqüencial igual ou menor a 5,1%. Esta fração pode ser considerada um ponto positivo de um ponto de vista de escalabilidade, pois, de acordo com o gráfico da figura 7.2, sistemas com esta fração paralelizável apresentam ganho significativo de desempenho quando escalados a até 1024 CPUs, e esta quantidade de CPUs não deve ser atingida em computadores multicore em menos de uma década.

7.2.6 Teste com Cargas Desbalanceadas

Com este teste, pretende-se verificar o efeito do uso de cargas desbalanceadas no desempenho de um sistema baseado no MTC. Os testes foram rodados duas vezes, uma sobre a equação A e outra sobre a equação B, sempre utilizando a máquina 2.

Para desbalancear o consumo de CPU das tarefas, foi adicionada ao programa, uma configuração que permite multiplicar o número de cálculos executados por cada tarefa individualmente de forma controlar por estas tarefas.

7.2.6.1 Passos

1. Para cada uma das equações A e B, configurar uma das tarefas para executar o dobro de cálculos do que as demais;
2. Invocar a aplicação exemplo;
3. Coletar o tempo de execução do sistema e de cada uma das tarefas;
4. Repetir o teste a partir do passo 1 escolhendo uma nova tarefa para incrementar o número de cálculos;

7.2.6.2 Resultados

Para a equação A, em cada execução do teste foi aumentado o número de cálculos realizados por uma tarefa escolhida ao acaso. Na tabela 7.8, a coluna Cálculos apresenta o total de cálculos realizados por cada tarefa. As tarefas são seis

no total, portanto, em todas as linhas onde o valor da coluna cálculos é um múltiplo de seis as tarefas estão perfeitamente balanceadas.

A coluna "Maior tarefa" apresenta o maior número de cálculos executados por uma única tarefa.

Tabela 7.8: Tempo total das execuções para a equação A.

Execução	Tempo total (s)	Cálculos	Maior tarefa
1	5,17	6	1
2	9,26	7	2
3	9,40	8	2
4	9,48	9	2
5	9,61	10	2
6	9,61	11	2
7	9,63	12	2
8	13,97	13	3
9	14,35	18	3
10	18,88	19	4

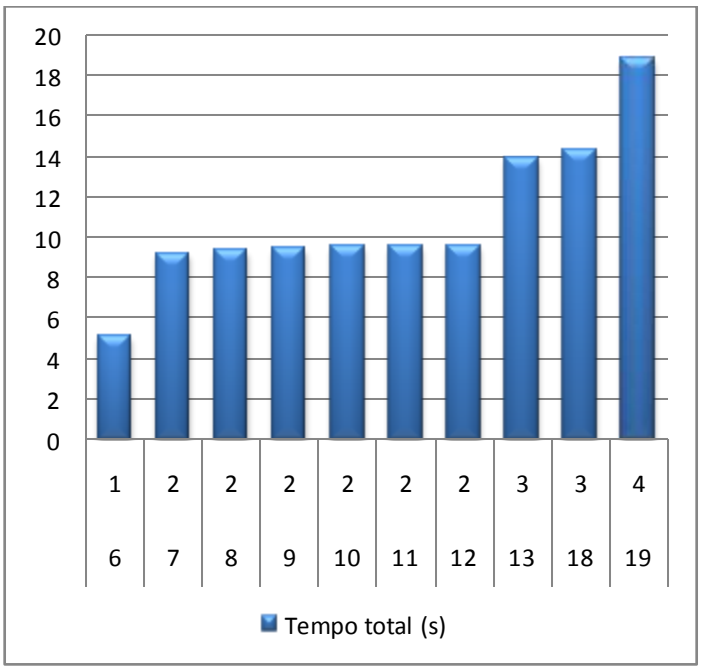


Figura 7.12: Tempo total de execução da equação A pelo sistema (y). No eixo x, dois valores distintos. Acima, o número de cálculos da maior tarefa, e abaixo, o número total de cálculos.

O teste foi executado de forma semelhante para a equação B. Nesta equação há 13 tarefas, portanto as execuções perfeitamente balanceadas são aquelas cujo total de cálculos é múltiplo de 13. Os resultados das execuções são apresentados a seguir.

Tabela 7.9: Tempo total das execuções para a equação B.

Execução	Tempo total (s)	Cálculos	Maior tarefa
1	8,41	13	1
2	9,98	15	2
3	14,04	17	2
4	14,18	19	2
5	14,39	21	2
6	15,35	23	2
7	16,73	25	2
8	17,37	26	2
9	16,98	27	3

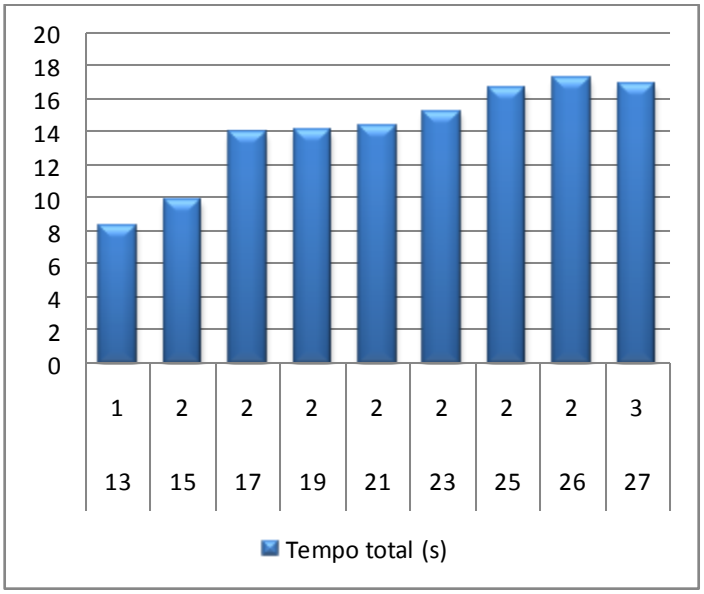


Figura 7.13: Tempo total de execução da equação B pelo sistema (y). No eixo x, dois valores distintos. Acima, o número de cálculos da maior tarefa, e abaixo, o número total de cálculos.

7.2.6.3 Análise

Em função dos resultados obtidos para a equação A, é possível concluir que o tempo de execução total do algoritmo está diretamente relacionado ao tamanho da maior tarefa, e não ao tamanho total do trabalho a ser realizado. Isto confirma o comportamento do paralelismo temporal dentro do sistema, onde o tempo total é proporcional à etapa mais lenta.

Os resultados obtidos para a equação B mostram uma situação bem diferente daquela obtida para a equação A. Neste caso, a cada incremento no número total de cálculos, há um aumento no tempo total de execução. A razão para isto é a existência de mais tarefas do que processadores disponíveis para resolver a equação

B. Neste caso, embora o desbalanceamento das tarefas crie um gargalo ao paralelismo temporal, o gargalo imposto pelo número de processadores disponíveis será o determinante até que seja superado.

7.3 Testes do sistema de controle autônomo de futebol de robôs

Para a verificação do sistema de controle, duas etapas de testes foram realizadas. Para sua execução, foi utilizada uma plataforma de futebol de robôs simulada, criada dentro deste trabalho.

A primeira etapa consiste na verificação funcional do sistema de controle, e visa constatar a viabilidade prática de se desenvolver uma aplicação real baseada no modelo de programação proposto. Os testes consistem na execução do programa em situações controladas de um jogo que permitem avaliar a autonomia do sistema e a efetividade das tarefas executadas.

A segunda etapa tem como objetivo, constatar o paralelismo dentro deste sistema de controle. Esta etapa de testes possui um objetivo ligeiramente diferente ao dos testes realizados na seção anterior. Antes, a intenção era verificar o paralelismo possível de ser obtido através do modelo de programação proposto. Agora, a intenção é verificar o paralelismo efetivo em uma aplicação real. O principal desafio para a realização desta etapa é a inexistência de um benchmark consolidado nesta área, dificultando o comparativo com outros trabalhos.

Os testes da segunda etapa avaliam o balanceamento das tarefas e traçam um comparativo da execução do sistema de controle em um número diferente de processadores.

7.3.1 Plataforma Para Testes Simulados

Para reduzir a complexidade inerente aos componentes motor e visual, bem como a dependência destes, foi projetado e desenvolvido um simulador de futebol de robôs levando em consideração as normas da liga F180 da Robocup.

Este simulador provê duas interfaces, ou *Application Progrraming Interfaces* (API), uma para entrada de dados, que corresponde ao componente motor, e uma para saída de dados, que corresponde ao componente visual.

A API motora transmite a um robô comandos de direção, velocidade bem como ativação e desativação do seu chutador e do seu driblador.

A API visual permite ler as informações de posicionamento físico dos robôs a partir do simulador.

A partir do simulador é permitido ao usuário visualizar da situação do jogo em tempo hábil e interagir com este para posicionar os robôs ou a bola em sua posição desejada dentro do campo.

Ao trabalhar com robôs reais, é necessário realizar-se uma série de ajustes finos, de forma a permitir que um robô desempenhe um comando adequadamente como esperado pelo sistema de controle. Para citar um exemplo, é necessário realizar o tratamento da diferença entre o comportamento de um robô com a bateria mais carregada comparado àquele de um robô com menor energia disponível, já que eles vão desempenhar um mesmo comando com velocidades diferentes.

Além disto, para que o sistema de controle seja capaz de obter um modelo lógico da situação de um jogo, portanto, tratável de um ponto de vista computacional, é necessário converter a imagem captada pela câmera de vídeo que monitora o jogo em uma série de estruturas de dados úteis. Esta conversão não é uma tarefa trivial, precisando ser constantemente ajustada para que apresente bom desempenho e um nível de ruído aceitável.

Estes dois problemas vão alem da responsabilidade do controle, sendo que geralmente é feita uma divisão do software para futebol de robôs em três partes principais, a parte motora, ou de locomoção, a parte visual, e a parte de controle, geralmente chamada de IA (inteligência artificial).

Figura 7.14: Principais componentes do ambiente simulado.

Ambas APIs introduzem um ruído gaussiano aos seus resultados de forma a aproximar os resultados daqueles obtidos em um ambiente real. Este ruído pode ter sua intensidade ajustada em função do teste que se deseja executar.

7.3.2 Teste 1: Verificação da efetividade do sistema de controle

Para verificar a efetividade das jogadas foram realizadas validações isoladas de cada um dos algoritmos de execução e posteriormente, o sistema de controle foi colocado em execução por 10 minutos e com isto verificada a sua autonomia. A seguir são apresentados alguns resultados destas validações por meio de diagramas. Cada diagrama apresenta o rastro dos objetos (robôs e bola) dentro do campo. A amostragem é realizada a cada 250 ms. As cruzes grandes representam a posição dos jogadores, a pequena corresponde à posição da bola.

7.3.2.1 Desviar obstáculos

Neste experimento um robô deve se deslocar de um ponto A até um ponto B desviando os obstáculos em seu caminho. Isto é feito realizando execução concorrente das ações “Evitar obstáculos” e “Andar em direção a um alvo”. A figura 7.15 mostra o resultado da sua execução, desenhando a trajetória do robô R2 por entre os obstáculos O1, O2 e O3 até atingir seu destino em B.

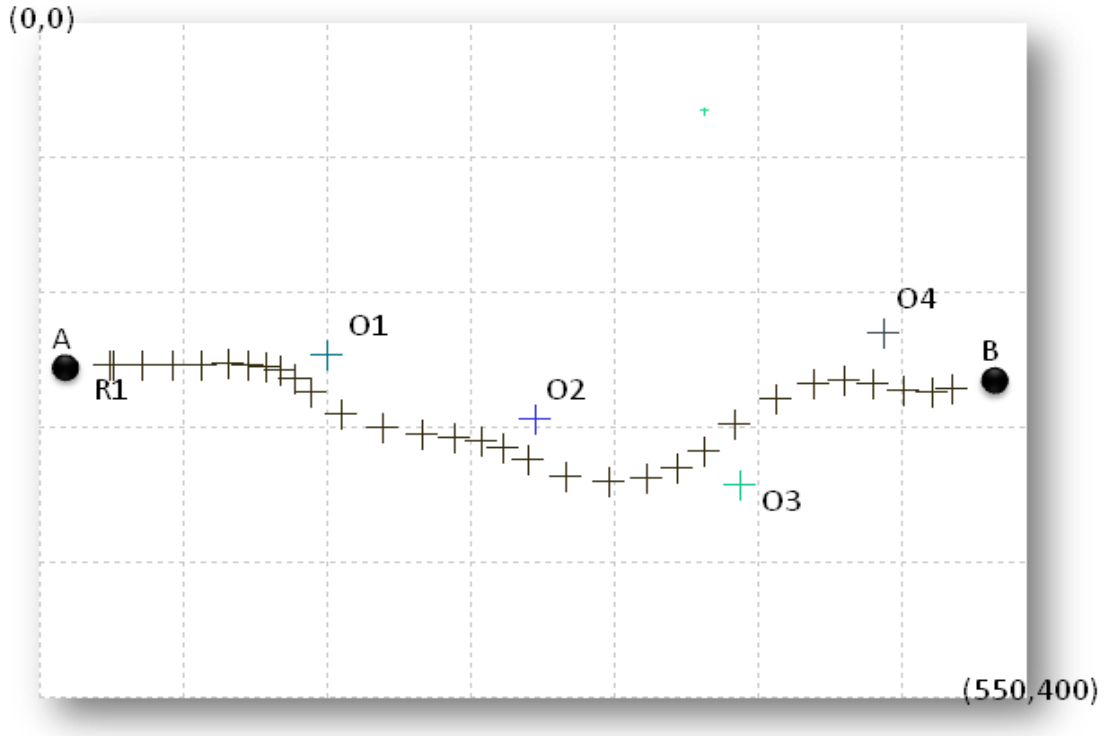


Figura 7.15: Resultado da execução concorrente das ações “Evitar obstáculos” e “Andar em direção a um alvo”. Os eixos x e y representam as dimensões do campo em centímetros.

7.3.2.2 Passar bola e receber um passe

Neste experimento um robô R1 deve se deslocar de um ponto A e passar a bola ao robô R2 que se desloca ao ponto B. Isto é obtido pela execução da jogada “Cruzamento”, que combina diversas ações como “Carregar a bola” e “Receber passe”. A figura 7.16 mostra o resultado da sua execução onde R1, R2, R3, R4 e R5 são os robôs de um time. A jogada mostra a trajetória percorrida por R1 até o ponto A, de onde chuta a bola em direção a R2. A trajetória de R2 mostra seu deslocamento ao ponto B em oposição ao movimento da bola, compensando a sua velocidade para receber adequadamente o passe.

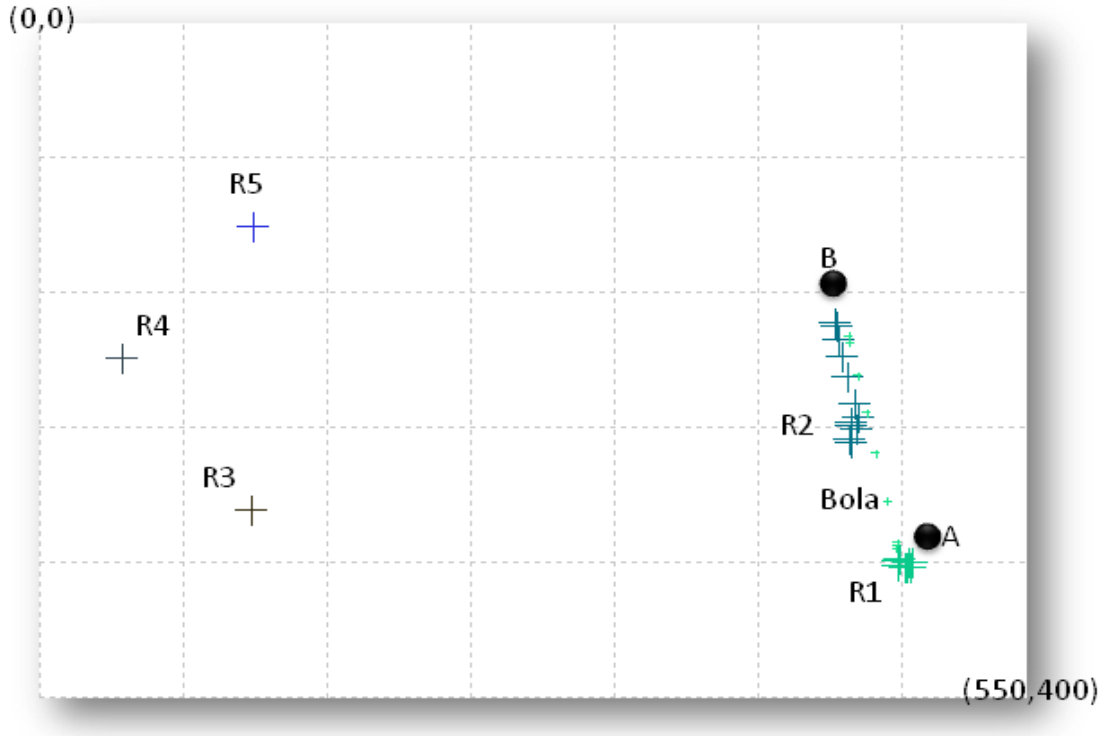


Figura 7.16: Resultado da execução da jogada “Cruzamento” envolvendo os robôs R1 e R2. Os eixos x e y representam as dimensões do campo em centímetros.

7.3.2.3 Posicionar os robôs

Neste experimento todos os robôs do time são posicionados para o inicio da partida. A execução da jogada “Posicionamento para iniciar jogo” realiza diversas ações de deslocamento em paralelo. A figura 7.17 mostra o resultado da sua execução, onde é vista a trajetória de cada robô deslocando-se a uma posição pré-estabelecida no campo.

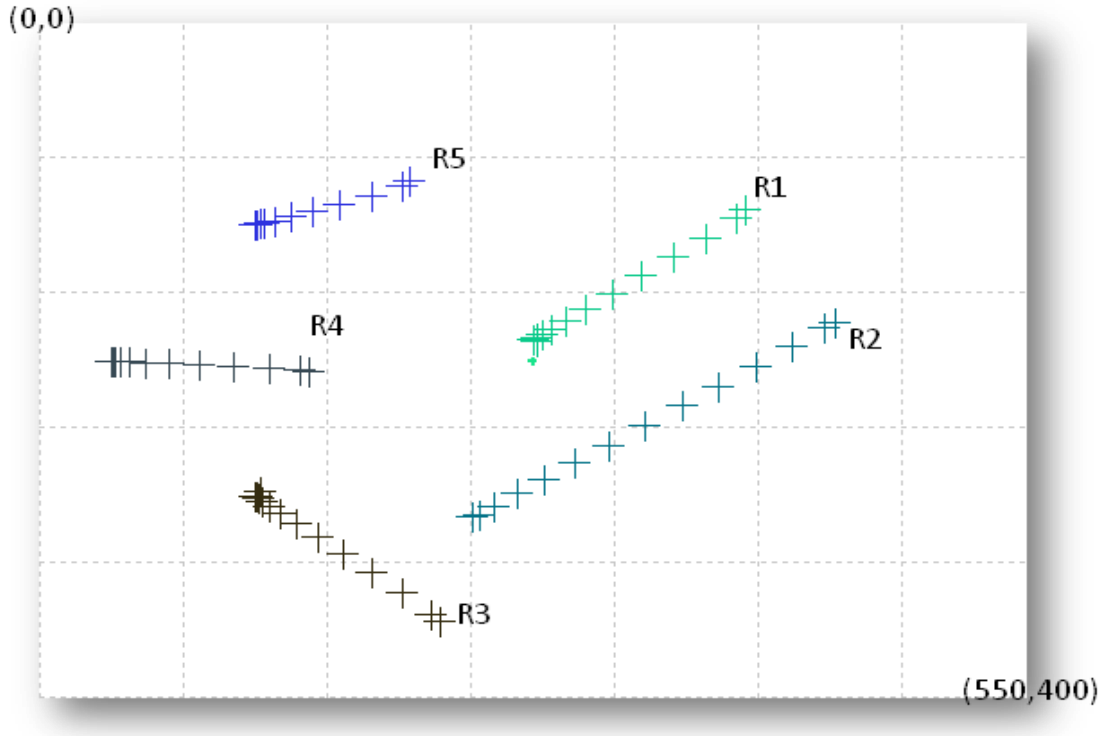


Figura 7.17: Resultado da execução da jogada “Posicionamento para iniciar jogo” envolvendo todos os robôs de um time. Os eixos x e y representam as dimensões do campo em centímetros.

7.3.3 Execução variando o número de processadores disponíveis

Este teste tem o objetivo de quantificar a importância da execução paralela para o sistema de controle desenvolvido. Para realizar esta validação, o sistema de controle teve seu consumo de processamento aumentado artificialmente de forma a exigir mais dos processadores disponíveis, tornando mais evidente as diferenças na execução com um diferente número de processadores. Para aumentar a carga do sistema, foram adicionados laços que forçam a tarefa a esperar por alguns milissegundos antes de prosseguir com seu processamento normal. Estes laços são invocados a cada estímulo recebido pela TC. Para estes testes não foi utilizado um time adversário, para evitar que a sua atuação influenciasse os resultados.

A execução dos testes foi feita na Máquina 2, utilizada para os experimentos anteriores, e foram realizados os passos descritos a seguir.

7.3.3.1 Passos

1. Iniciar o simulador colocando a bola ao centro do campo e os jogadores em suas posições iniciais;
2. Definir a afinidade do processo habilitando 1 CPU;
3. Executar o sistema de controle por no máximo 30 segundos;
4. Verificar se o time realizou um gol no time adversário anotando o tempo decorrido desde o inicio da execução;
5. Realizar os passos anteriores novamente totalizando 8 execuções;

6. Aumentar o número de CPUs habilitadas para o processo em 1 e repetir os passos até que se atinja o número de 8 CPUs;

7.3.3.2 Resultados

Como resultado da execução destes testes, foi coletada a taxa de sucesso do sistema de controle em realizar um gol de forma autônoma variando-se o número de CPUs disponíveis, foi também coletado o tempo médio consumido para que um gol fosse realizado, como mostrado pelas tabelas e gráficos a seguir.

Tabela 7.10: Tempo médio de execução de cada um dos algoritmos.

CPUs	Taxa de Sucesso	Tempo médio (s)
1	0,00%	0,00
2	41,67%	12,90
3	75,00%	10,33
4	87,50%	11,00
5	100,00%	10,88
6	100,00%	10,25
7	100,00%	10,25
8	100,00%	9,63

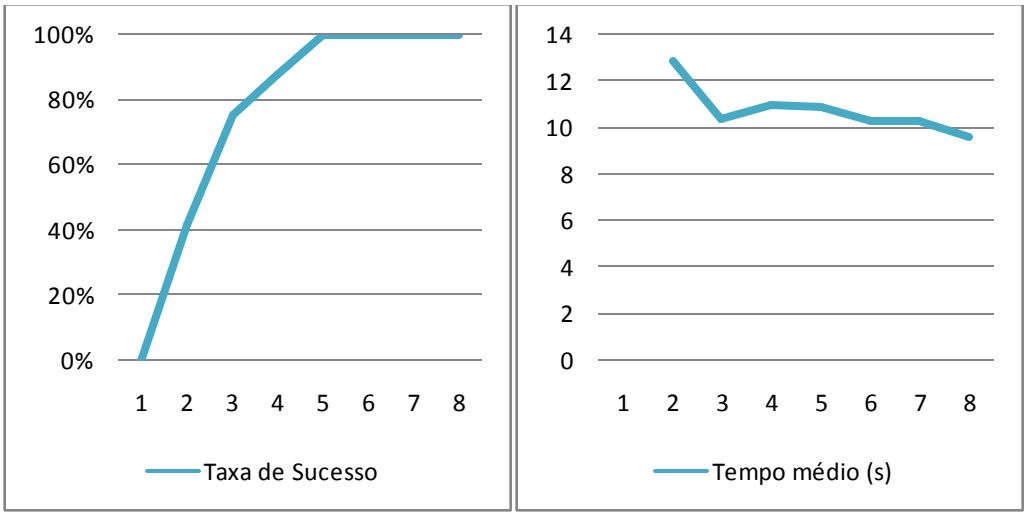


Figura 7.18: Taxa de sucesso e o tempo médio consumido para realizar um gol em função do número de processadores.

7.3.3.3 Análise

Em função dos resultados obtidos, foi possível verificar que o número de processadores influencia diretamente a capacidade do sistema de controle realizar suas tarefas. Com apenas um processador, o tempo de resposta do sistema a mudanças no ambiente tornou-se muito alto, impactando de tal forma a precisão nos movimentos realizados pelos jogadores que nenhum gol foi realizado. À medida que mais processadores foram disponibilizados, a taxa de sucesso aumentou gradativamente até chegar a 100%, onde todas as jogadas realizadas pelo sistema resultaram em gols, evidenciando o paralelismo de tarefas dentro do sistema.

A partir de quatro processadores, a taxa de sucesso atingiu seu máximo, permanecendo inalterada com o aumento do total de processadores disponíveis. Para que fosse possível verificar uma evolução contínua do sistema em função do número de processadores, seria necessário ter atingido o limite de processamento disponível na máquina utilizada, o que não ocorreu.

Cabe ressaltar que o desempenho do sistema foi artificialmente degradado para que o paralelismo ficasse mais evidente. Em cada estímulo recebido por uma tarefa, uma espera de 50 ms foi adicionada para consumir mais capacidade de processamento. Não foi possível atingir o limite da máquina utilizada devido à natureza do problema sendo resolvido. Por se tratar de um sistema de tempo hábil, a adição de laços para consumir tempo do processador torna o tempo de resposta do sistema mais lento, e após ter sido atingido o limite de 50 ms, este intervalo inviabiliza o funcionamento correto do sistema pois acrescenta tempo ao seu caminho crítico (maior fração seqüencial). Outra possibilidade seria a inclusão de novas tarefas fora do caminho crítico do sistema com o objetivo de onerar os processadores, porém isto iria contra o objetivo deste teste que é avaliar a conquista de paralelismo dentro das tarefas necessárias.

Dentro destes testes, um segundo indicador de desempenho foi utilizado: o tempo necessário para realizar um gol. Computando a sua média, foi possível verificar uma tendência de diminuição deste tempo à medida que mais processadores foram disponibilizados para o sistema. Isto se deve à maior precisão nos movimentos dos robôs em função de respostas mais velozes a mudanças no ambiente. O que esta análise apresenta é que, na grande parte do tempo de execução do sistema, não existem tarefas ocorrendo em paralelo que demandem mais de quatro processadores, porém em alguns momentos específicos, a presença de mais processadores constituiu diferença no tempo de resposta do sistema.

Uma última análise permite concluir que embora o sistema desenvolvido seja leve em termos de necessidade de tempo de processamento, pois seus algoritmos consomem apenas frações de microssegundos para receber um estímulo de entrada e transformá-lo em estímulos de saída, a sua arquitetura abre as portas para que algoritmos de execução muito mais complexos, da ordem de milissegundos, sejam utilizados, atingindo o verdadeiro objetivo por trás deste trabalho: viabilizar o uso de técnicas elaboradas para o controle de futebol de robôs.

8 CONCLUSAO E TRABALHOS FUTUROS

Através desta dissertação foi descrito um modelo de programação paralela inspirado no modelo geral do conexionismo, validado pela sua aplicação em um sistema de controle autônomo para futebol de robôs, atingindo assim, os dois principais objetivos propostos neste trabalho.

Foi possível concluir que é aceitável adaptar um sistema paralelo que utiliza coordenação baseada em eventos dentro do modelo geral proposto por Rumelhart, McClelland e Hinton (MCCLELLAND, RUMELHART e HINTON, 1983) e com isto, produzir um modelo de programação que herda características próprias dos sistemas inteligentes biológicos para solução de problemas de tempo hábil. Além disto, foi possível determinar a viabilidade do projeto e desenvolvimento de um sistema correto, de um ponto de vista funcional, e fiel a este modelo. Através de experimentos foi demonstrado o potencial do modelo desenvolvido para o ganho de desempenho por meio do paralelismo, com evolução próxima a linear do *speedup* e uma fração seqüencial igual ou inferior a 5,1% em seus mecanismos internos.

É possível verificar que todos os conceitos de um sistema conexionista permanecem válidos em diferentes escalas, pois, embora o modelo proposto reduza a escala de uma rede conexionista, onde em cada unidade seja representada uma infinidade de unidades de uma rede tradicional, todos os componentes determinantes de um sistema conexionista foram colocados em prática com sucesso. Enquanto alguns dos conceitos se tornam mais simples pela possibilidade de se entender o valor funcional de cada unidade, alguns destes conceitos introduzem desafios próprios quando observados por uma escala menor, podendo-se citar como exemplo o mecanismo de aprendizado, que em uma escala aumentada requer pouca preocupação do desenvolvedor com relação à topologia e aos dados trafegados entre as unidades, e em uma escala reduzida exige uma grande quantidade de trabalho de projeto para que se possa produzir uma rede com potencial para treinamento.

Foi possível, também, desenvolver um sistema de controle autônomo para futebol de robôs que tira proveito do processamento paralelo, potencializando avanços na busca por vantagens competitivas na era dos computadores multicore. Neste sistema, foram identificadas técnicas de inteligência artificial para controle de robôs móveis completamente adequadas à arquitetura paralela utilizada, como por exemplo, campos potenciais e lógica *fuzzy*. A possibilidade do uso destas técnicas consolidadas aproveitando os ganhos de desempenho provenientes da execução concorrente vem a demonstrar a flexibilidade do sistema desenvolvido, sendo um elemento chave para que este sistema possa evoluir a ponto de ser

utilizado em competições. O sistema de controle foi submetido a testes que mostraram o uso efetivo de paralelismo em seus algoritmos disponibilizando ao programador uma quantidade muito maior de tempo de processamento do que seria possível pelo uso de um algoritmo seqüencial, chegando, em alguns casos, à faixa dos milissegundos.

Além das conclusões diretamente relacionadas aos objetivos do trabalho, também é apropriado salientar que a área de processamento paralelo vem adquirindo cada vez mais importância dentro da ciência da computação, em função dos avanços dos computadores com múltiplos processadores. Isto está gerando um movimento irreversível da área de desenvolvimento de software na busca por soluções apropriadas para o aproveitamento de um tipo de desempenho que há pouco tempo não era imaginado pelos desenvolvedores e rapidamente irá se tornar um pré-requisito para a construção de novos sistemas.

Em função desta revolução, é possível perceber que, aos poucos, os conceitos dentro do processamento paralelo vão ficando mais claros e simples de aplicar. Enquanto alguns livros mais antigos como (PARHAMI, 1999) ainda apresentam noções bastante confusas quanto à classificação das máquinas paralelas e das ferramentas de desenvolvimento disponíveis, literaturas mais recentes como (MATTSON, SANDERS e MASSINGILL, 2004) e (LEISERSON e MYRMAN, 2008) já apresentam definições mais objetivas e compreensíveis.

Esta evolução se reflete também nas ferramentas de software, onde mecanismos como OpenMP, MPI e linguagens com suporte nativo à concorrência se apresentam como sobreviventes de uma infinidade de plataformas de concorrência e comunicação inter-processos existentes há menos de uma década. Com isto, passou a ser possível às pesquisas em software paralelo focarem nos demais problemas da execução concorrente, simplificando cada vez mais o desenvolvimento de software paralelo.

8.1 Trabalhos Futuros

Alguns pontos específicos dentro dos objetivos propostos não puderam ser completamente abordados e permanecem como tema para trabalhos futuros.

Primeiramente, dentro da área de interesse do trabalho, é preciso finalizar o sistema de controle autônomo com uma gama maior de jogadas, oferecendo um nível mais elevado de autonomia, e por fim levá-lo da plataforma simulada para a plataforma real, onde será possível compará-lo às abordagens existentes por meio de competições. Ainda dentro desta área, se deseja utilizar o mecanismo de aprendizado do MTC para ajuste dos parâmetros das jogadas tornando o sistema mais preciso em sua atuação.

Outro ponto em que foram encontradas dificuldades foi a falta de um benchmark consolidado para futebol de robôs dentro da liga SSL. A busca por um padrão de desempenho que possa ser utilizado como benchmark por trabalhos semelhantes é um trabalho importante que pode ser continuado a partir deste. Para atingir este objetivo é necessária a consolidação de uma plataforma padrão para testes simulados e a publicação de um sistema de controle que possa ser utilizado como base de comparação para outras equipes. Ambas as metas podem ser facilitadas pelo trabalho já desenvolvido através da divulgação da plataforma de

testes e do sistema de controle em projetos de código aberto dentro da comunidade interessada.

Por fim, o modelo de programação produzido ainda requer bastante esforço do programador para a produção e avaliação de um sistema que tire proveito de todas as suas características, e em especial o aprendizado. A continuidade do trabalho com o objetivo de simplificar este modelo ampliaria a sua aplicabilidade a uma gama maior de problemas.

9 REFERENCIAS BIBLIOGRAFICAS

ABBOTT, B.; NEEMA, S. **Electrotechnical Commission (IEC) document 61131-7**. Las Vegas: Int. Conf. on Parallel and Distributed Processing Techniques and Applications. 1997.

AKENINE-MÖLLER, T. **Real-time rendering**. 3a Edição. ed. Natik: A. K. Peters Ltd., 2008.

ATTIYA, H. Concurrency and the Principle of Data Locality. **IEEE Distributed Systems Online**, v. 8, 2007. ISSN 1541-4922.

BARNEY, B. Lawrence Livermore National Laboratory. **Introduction to Parallel Computing**, 25 maio 2010. ISSN UCRL-MI-133316. Disponivel em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: 14 set. 2010.

BERNACKI, M.; WLODARCZYK, P. Principles of training multi-layer neural network using backpropagation. **Site do Virtual Laboratory of Artificial Intelligence da AGH - UST**, 06 set. 2004. Disponivel em: <http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html>. Acesso em: 13 out. 2010.

BOER, R. D.; KOK, J. **The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team**. Universiteit van Amsterdam. Amsterdam. 2002.

BUNTINAS, D.; MERCIER, G.; GROPP, W. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. **Parallel Comput.**, v. 33, p. 634--644, 2007. ISSN 0167-8191.

BUSI, N.; GORRIERI, R.; ZAVATTARO, G. **On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages**. [S.l.]. 2000.

DI GIUSTO, C.; GABBRIELLI, M. **Full abstraction for Linda**. [S.l.]: Springer-Verlag. 2008. p. 78--92.

DONGARRA, J. J.; OTTO, S. W.; SNIR, M. **A message passing standard for MPP and workstations**. Communications of the ACM. [S.l.]: [s.n.]. 1996. p. 84–90.

DORAI, G. K.; YEUNG, D.; CHOI, S. **Optimizing smt processors for high single-thread performance**. [S.l.]: [s.n.]. 2003. p. 1--35.

EAGER, D. L.; ZAHORJAN, J.; LAZOWSKA, E. D. **Speedup versus efficiency in parallel systems**. Computers, IEEE Transactions on. [S.l.]: [s.n.]. 1989. p. 408-423.

FLUET, M. et al. **Manticore:** a heterogeneous parallel language. [S.l.]: ACM. 2007. p. 37--44.

FLYNN, M. J. **Some Computer Organizations and Their Effectiveness**. IEEE Trans. Comput., Vol. C-21. [S.l.]: [s.n.]. 1972. p. 988.

FOSTER, I. **Designing and Building Parallel Programs:** Concepts and Tools for Parallel Software Engineering. Boston: Addison-Wesley Longman Publishing Co., Inc., 1995.

GAMMA, E. **Design patterns:** elements of reusable object-oriented software. [S.l.]: Pearson Education, 2003.

GODFREY, P. B.; KARP, R. M. **On the price of heterogeneity in parallel systems**. [S.l.]: ACM. 2006. p. 84--92.

HAYASHI, Y.; SUEHIRO, K. **Hybrid parallelization and flat parallelization in HPF (high performance Fortran)**. [S.l.]: Springer-Verlag. 2008. p. 305--314.

HAYKIN, S. **Neural Networks:** A Comprehensive Foundation. [S.l.]: Prentice Hall, 1999.

HENNESSY, J. L.; PATTERSON, D. A. **Organização e projeto de computadores**. [S.l.]: Morgan Kaufmann Publishers Inc. 2000.

HILLIS, W. D.; STEELE, J. Data parallel algorithms. **Commun. ACM**, v. 29, p. 1170--1183, 1986. ISSN 0001-0782.

INTEL. Intel Tera-scale Research Program. **Intel Tera-scale Research Program**, 2010. Disponivel em: <http://techresearch.intel.com/articles/Tera-Scale/1421.htm>. Acesso em: 10 maio 2010.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO 8373:1994**. International Organization for Standardization. [S.l.]. 1994.

JACAK, W. **Intelligent Robotic Systems:** Design, Planning, and Control. [S.l.]: Perseus Publishing, 1999.

KIRKWOOD-WATTS, C. R. **Cooperation in dynamic teams of mobile robots**. [S.l.]. 2000.

KLAIBER, A. C.; LEVY, H. M. A Comparison of Message Passing and Shared Memory Architectures. **IEEE**, 1994. 12.

KOSTIADIS, K.; HU, H. **A Multi-threaded Approach to Simulated Soccer Agents for the RoboCup Competition**. [S.l.]: Springer-Verlag. 2000. p. 366--377.

KRIENGWATTANAKUL, A. Plasma-z 2008 Team Description Paper. **Robocup**, 2008.

LEISERSON, C. E.; MYRMAN, I. B. **How to survive the multicore software revolution**. Rev. R21.3. ed. [S.l.]: CILK Arts, 2008.

LOBB, C. J. et al. **Parallel Event-Driven Neural Network Simulations Using the Hodgkin-Huxley Neuron Model," presented at**. [S.l.]: [s.n.]. 2005.

MATTSON, T.; SANDERS, B.; MASSINGILL, B. **Patterns for parallel programming**. [S.l.]: Addison-Wesley Professional, 2004.

MCCLELLAND, J. L.; RUMELHART, D. E.; HINTON, G. E. **The Parallel Distributed Processing Perspective**. [S.l.]: [s.n.], 1983.

MORAVEC, H. When will computer hardware match the human brain? **Journal of Evolution and Technology**, dez. 1998. 12.

MOURA E SILVA, L.; BUYYA, R. **Parallel Programming Models and Paradigms**. [S.l.]: Prentice Hall, 1999.

OAKS, S.; WONG, H. **Java Threads**. 3a Edição. ed. [S.l.]: O'Reilly Media Inc., 2004.

PARHAMI, B. **Introduction to Parallel Processing:** Algorithms and Architectures. [S.l.]: Kluwer Academic Publishers, 1999.

RAUBER, T.; RÜNGER, G. **Parallel Programming for Multicore and Cluster Systems**. [S.l.]: Springer Berlin Heidelberg, 2010.

REINDERS, J. **Intel threading building blocks**. [S.l.]: OReilly & Associates, Inc., 2007.

ROBOCUP. **Robocup.org**, 2010. Disponivel em: <http://www.robocup.org/>. Acesso em: 11 maio 2010.

SIEGWART, R.; NOURBAKHSH, I. R. **Introduction to Autonomous Mobile Robots**. [S.l.]: Bradford Company, 2004.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2a Edição. ed. São Paulo: Prentice Hall, 2003.

TAYLOR, S. et al. The Concurrent Graph: Basic Technology for Irregular Problems. **IEEE Parallel Distrib. Technol.**, v. 4, p. 15--25, 1996. ISSN 1063-6552.

TORRES, E.; WEITZENFELD, A. **RoboCup Small-Size League:** Using Neural Networks to Learn Color Segmentation during Visual Processing. IEEE Latin American Robotic Symposium. [S.l.]: [s.n.]. 2008. p. 14-19.

WELC, A.; JAGANNATHAN, S.; HOSKING, A. **Safe futures for Java**. [S.l.]: Press. 2005. p. 439--453.

WELCH, G.; BISHOP, G. An Introduction to Kalman Filter. **SIGGRAPH**, 24 jul. 2006.

ZICKLER, S. et al. **CMDragons Extended Team Description**. [S.l.]: [s.n.]. 2009.