Escalonamento em Sistemas Paralelos e Distribuídos

Prof. Adenauer Corrêa Yamin

Universidade Católica de Pelotas e Universidade Federal de Pelotas
E-mail: adenauer{@atlas.ucpel.tche.br; @ufpel.tche.br}
http://www.ucpel.tche.br e http://www.ufpel.tche.br

O Contexto da Área de Escalonamento

Após um programa ter sido projetado e construído como um conjunto de tarefas que interagem, uma alocação destas nos elementos processadores de uma arquitetura paralela precisa ser determinada. Este problema de alocação é genericamente chamado de "escalonamento".

O escalonamento de tarefas paralelizáveis se mostra computacionalmente intratável ainda em muitos casos. Por outro lado, um número cada vez maior de tipos de arquiteturas paralelas e distribuídas tem sido colocadas no mercado a cada ano, todas contemplando especificidades para o escalonamento de tarefas nos seus recursos computacionais. Um estudo neste sentido foi feito no trabalho [95], onde, dentre outros aspectos, ficou evidenciada a heterogeneidade entre os diversos equipamentos disponibilizados ao usuário final, bem como o curto espaço de tempo entre as gerações de hardware de um mesmo fabricante.

Isto leva a uma constante reavaliação, tanto do software básico (escalonador, etc.) como do software aplicativo, seja para obter maior eficiência, seja para viabilizar aplicações cujo grão computacional ou de comunicações conduziam a um desempenho insatisfatório na geração anterior do equipamento.

Finalmente, é preciso ter presente a premissa de exploração implícita do paralelismo atualmente buscada por diversos grupos de pesquisa (vide projeto APPELO - [40]). Esta forma de explorar o paralelismo, transfere do usuário para o ambiente de execução, toda uma complexidade de procedimentos.

Por tudo isto, a pesquisa de estratégias de escalonamento vem se mantendo um dos mais desafiadores problemas em computação paralela e distribuída.

Escalonamento de Tarefas: Um Problema Clássico

O problema do escalonamento de tarefas tem sido abordado de diversas maneiras, nas diferentes áreas da atividade humana: sistemas de produção em indústrias, diferentes sistemas de atendimento a usuários, serviços de transporte (aéreo, ferroviário, marítimo, etc.). Como já introduzido na seção 0, particularmente neste curso, vão ser tratados os aspectos pertinentes ao escalonamento das aplicações que são submetidas a uma arquitetura paralela/distribuída para execução.

Os conceitos, nesta seção, são baseados em [27] e [88]. Os mesmos são gerais, e serão apresentados de forma resumida, com o intuito apenas de contextualizar o tema. Ao longo do curso, de forma recursiva, serão explorados os conceitos correspondentes a cada aspecto que vier a ser tratado.

Visão Geral (informal)

De modo geral, o problema do escalonamento contempla um conjunto de recursos e outro de consumidores. Baseado na natureza e nas restrições, tanto dos recursos como dos consumidores, o problema consiste em encontrar uma política eficiente para gerenciar o uso dos recursos pelos vários consumidores, de forma que seja otimizada a medida de desempenho tida como parâmetro. A figura 38 retrata esta situação.

Figura 38: O Problema do Escalonamento

Usualmente, uma proposta de escalonamento é avaliada por duas características: desempenho e eficiência. Deste modo, a avaliação abrange tanto o escalonamento obtido como a própria política empregada pelo escalonador. Por exemplo, se o critério para julgar o escalonamento gerado for o tempo que um programa paralelo necessita para sua execução, quanto menor este, melhor o escalonador. Por sua vez, a política de escalonamento (utilizada pelo escalonador) pode ser avaliada em função da sua complexidade computacional. Se duas políticas produzem escalonamentos iguais, a mais simples (menor custo computacional) é seguramente a melhor ([27]).

Visão Semi-Formal (mais precisa)

Um escalonamento é uma função σ que associa cada tarefa (processo) tanto com uma data na qual sua execução irá começar, como com um processador específico da arquitetura. Assim, um escalonamento do conjunto de tarefas $V(v)$ com suas respectivas precedências $E(e)$ - $G = (V, E)$, sobre os m processadores da arquitetura é um par $\sigma = (data, aloc)$ de funções $data : V \to N$ e $aloc : V \to \{1,2,...,m\}$, as quais assinalam, para cada tarefa, a data (momento de início) da execução e o processador que vai executá-la, respectivamente ([88]).

Um escalonamento $\sigma = (data, aloc)$ é valido se, a cada momento, os m processadores estão participando da computação, e todas as restrições de precedência são respeitadas, isto é:

$$\forall (T, T') \in E : data(T') \geq data(T) + exec(T) + Com(T, T').$$

Onde:

$exec(T)$ - é a função correspondente ao tempo de execução da tarefa

$data(T)$ - momento que inicia a execução da tarefa T

$Com(T, T') = 0$ se $aloc(T) = aloc(T')$ - o custo de comunicação usualmente é considerado nulo quando as tarefas têm a mesma alocação

O tempo total de execução C de uma aplicação paralela é definido como

$$C = Max_{T \in V} \left(data(T) + exec(T) \right)$$

O objetivo principal de qualquer escalonador é encontrar um escalonamento válido que minimize C.

Taxonomias para Área de Escalonamento em Arquiteturas Paralelas e Distribuídas

O objetivo desta seção é apresentar as relações entre as frentes de trabalho na área de escalonamento, e deste modo contextualizar os aspectos tratados neste curso. Por outro lado, o uso de taxonomias também objetiva minimizar a proliferação de terminologias e conceitos muitas vezes contraditórios.

Neste sentido, esta seção vai discorrer sobre duas entre as mais difundidas taxonomias em escalonamento de sistemas paralelos e distribuídos: a taxonomia de

Casavant e Kuhl [13], cujo foco é a natureza do algoritmo de escalonamento utilizado, e a taxonomia de Feitelson ([35], [34]), cuja base é a forma como os recursos da arquitetura paralela e/ou distribuída são compartilhados.

A busca de taxonomias para organizar a produção científica na área de escalonamento em sistemas paralelos e distribuídos não é uma preocupação nova. Diversos outros trabalhos menos conhecidos a respeito foram feitos. Mauny ([57]) organizou de forma abrangente a alocação de recursos em geral (memória, processadores, etc.) e sua relação com o modelo de programação e a arquitetura do sistema. Wang e Morris ([92]) organizaram sua proposta de taxonomia em função do grau de informações de dependência envolvidas na alocação de tarefas. Além destes trabalhos, outros com maior similaridade aos que serão explorados nesta seção também foram realizados. Um exemplo seria a proposta de El-Rewini apresentada em [27], a qual é bastante próxima à taxonomia de Casavant e Kuhl.

Considerações sobre a Diversidade das Propostas de Escalonamento

A figura 39 resume os contrastes entre os esquemas de escalonamento que têm sido propostos e implementados. Apesar de todos os esquemas tratarem do problema genérico: “como escalonar aplicações paralelas em uma arquitetura paralela”, as considerações específicas de cada caso fazem as instâncias deste problema bastante diferentes entre si ([35], [36]).

Uma diferença conceitual básica, por trás da diversidade, está em como ocorre a utilização do sistema. Muitas propostas têm por base sistemas *off-line*, nos quais é possível saber com antecipação as características da demanda. Por outro lado, está crescendo o emprego de sistemas *on-line*, nos quais a chegada de novas tarefas é imprevisível.

Outra fonte concreta para a diversidade das estratégias de escalonamento diz respeito às características do sistema paralelo. Em função destas características decorrem diferentes possibilidades a serem suportadas pelo software básico do equipamento paralelo (migração, preempção, troca de contexto, etc.), bem como a quantidade de informação que este disponibiliza para o escalonador ([36]). Outrossim, varia entre os tipos de arquiteturas, os custos de comunicação, poder de processamento/memória dos nodos, dentre outros aspectos.

Por fim, outra fonte de divergência para as estratégias de escalonamento, diz respeito às possíveis características das aplicações. Em [35] e [33] são feitos estudos neste sentido, os quais estão resumidos a seguir:

Natureza das Aplicações Paralelas

Rígidas: são aplicações que exigem um certo número pré-definido de processadores. Elas não irão processar em um número menor, e não utilizarão mais processadores do que o previsto.

Moldáveis: estas aplicações facultam que o número de processadores a ser utilizado possa ser definido externamente. Deste modo, se a aplicação for submetida à execução em um momento de elevada demanda, poderá ter o número inicialmente previsto de processadores reduzido pelo sistema. O número de processadores alocados é mantido durante toda execução.

Evolutivas: são as aplicações caracterizadas por uma execução com alternância de fases paralelas e seriais. Ao início de cada fase, a aplicação solicita do sistema os recursos de que irá necessitar durante a mesma, e ao final libera os mesmos. Este comportamento faculta ao escalonador otimizar a utilização de processadores enquanto a execução da aplicação evolui.

Maleáveis: são aplicações que, durante a execução, podem se ajustar às modificações promovidas pelo escalonador. Deste modo, o escalonador tem a liberdade de alterar o número de processadores utilizados pela aplicação em qualquer fase da execução. Quando algum processador se torna disponível, este pode ser imediatamente alocado a alguma das tarefas em execução no momento, reduzindo deste modo a possibilidade de fragmentação. Por sua vez, quando da chegada de novas aplicações, as mesmas podem ter sua execução iniciada imediatamente, utilizando processadores solicitados de outras aplicações.

A Taxonomia de Feitelson

Multiprogramação é a técnica de executar múltiplas tarefas concorrentemente no mesmo equipamento. Publicações sobre sistemas operacionais introduzem a multiprogramação como uma solução para a baixa utilização de recursos. Nesta técnica, uma adequada combinação de tarefas com exigências complementares pode manter melhor ocupadas as diferentes partes de um sistema computacional ([2]).

A mesma consideração pode ser aplicada aos sistemas paralelos. Novamente se faz necessária uma combinação adequada de tarefas, com a consideração adicional de que as tarefas que concorrem aos mesmos recursos tenham diferentes graus de paralelismo. Trabalhos neste sentido já existem há algum tempo ([52], [31], [89], [30]).

A importância da multiprogramação é clara, e seu uso para melhorar a resposta do sistema às solicitações dos usuários se mostra crucial nas arquiteturas paralelas, seja para justificar seu elevado custo, seja pelo objetivo central do processamento em elevado desempenho, que é retornar resultados ao usuário na maior brevidade possível (observar que *Ttotal* = *Tprocessamento* + *Tesperando_equipamento*).

Escalonamento em Um e Dois níveis

No escalonamento em Um-Nível, a ação de alocar recursos de processamento é combinada com a de decidir quais processos da aplicação irão utilizar cada um destes recursos. No escalonamento em Dois-Níveis, estes dois aspectos são desacoplados: um primeiro nível trata da alocação dos recursos para as aplicações e um segundo de que forma se dará o seu uso.

Escalonamento em Um-Nível

O inconveniente com o escalonamento em Um-Nível é o risco de que o sistema operacional do equipamento não responda satisfatoriamente às particularidades das diferentes aplicações. É importante ter presente que muitas das decisões de escalonamento são conseqüência das condições de sincronização/comunicação entre os processos da aplicação paralela. Por exemplo, é usual um processo ficar bloqueado aguardando outro enviar uma mensagem ou atingir um ponto de sincronização. Em aplicações de pequena granulosidade, tais interações são numerosas e ocorrem em taxas bastante elevadas. O fato do sistema operacional do equipamento arcar com os custos inerentes a cada uma destas situações de sincronização pode introduzir um *overhead* global proibitivamente elevado. Some-se a isto que, via de regra, o sistema operacional, em princípio, não está comprometido com as características de nenhum tipo de aplicação em particular, e por isto não tem como otimizar o seu escalonamento.

Contudo, existem situações que o escalonamento em Um-Nível é utilizado com sucesso. A seguir estão três casos típicos:

- alocação de processadores (Particionamento): será analisada na seção 0;
- compartilhamento de tempo: uma revisão sobre os principais mecanismos é feita na seção 0;
- escalonamento de grupos (*Gang Scheduling*): vide item 0.

Escalonamento em Dois-Níveis

No escalonamento em Dois-Níveis, o sistema operacional somente é responsável por alocar os recursos computacionais para a aplicação (particionamento - **primeiro nível**). A aplicação por si mesma, ou o seu ambiente de execução, realiza o escalonamento dos processos que a compõem, sobre os processadores para ela alocados. Este nível (**segundo**) de escalonamento interno, específico e comprometido com as aplicações, pode otimizar as condições de sincronização e comunicação.

			Tempo compartilhado			
			Sim			Não
			Processadores Independentes		Gang Scheduling	
			Fila Global	Filas Locais		
Espaço Compartilhado	Sim	Flexível	Mach	Paragon/service Meiko/timeshare KSR/interactive Transputers Tera/streams Chrysalis	Medusa Butterfly@LLNL Cray T3E Meiko/gang Paragon/gang SGI/gang Tera/PB MAXI/Gang	IBM SP2, Victor Meiko/batch Paragon/slice KSR/batch 2-Level/bottom TRAC, MICROS Amoeba
		Estruturado		NX/2 on IPSC/2 NCUBE	CM-5 Cedar DHC on SP2 DQT on RWC-1	Cray T3D CM-2 PASM Hypercubes
	Não		IRIX on SGI NYU Ultra Dynix 2-Level/top Hydra/C.mmp	StarOs Psyche Elxsi AP1000	MasPar MP2 Alliant FX/8 Chagori on K2	Illiac IV MPP GF11 Warp

Figura 39: A Taxonomia de Feitelson Aplicada

O Escalonamento em Dois Níveis: As Possibilidades de Compartilhamento

Conforme introduzido nas seções anteriores, a classificação proposta por Feitelson ([35]) é baseada na forma como os recursos computacionais são compartilhados: compartilhamento temporal, espacial ou ambos. A figura 39 apresenta uma taxonomia baseada nestes aspectos.

Um dos principais aspectos a serem observados, é que os mecanismos de compartilhamento de tempo são independentes daqueles para compartilhamento de espaço. Daí a organização da figura 39 ser em duas dimensões. Deste modo, uma

partição pode estar sendo compartilhada no tempo por diversas aplicações, enquanto outra não.

Em se tratando de compartilhamento de tempo, a distinção ocorre entre os mecanismos de escalonamento que trabalham cada processador individualmente e aqueles que gerenciam grupos de processadores como uma unidade lógica (*Gang Scheduling* - escalonamento preemptivo de um determinado grupo de processos - vide seção 0).

A Taxonomia de Casavant

Na sua taxonomia, originalmente publicada em [13], Casavant utilizou uma proposta híbrida: uma abordagem hierárquica e outra horizontal.

Classificação Hierárquica

A estrutura da classificação hierárquica pode ser vista na figura 40, e sua discussão está sintetizada a seguir.

Local & Global

Na taxonomia de Casavant este é o nível mais alto da hierarquia. **Escalonamento local** diz respeito às metodologias de compartilhamento de tempo (*time-sharing*), utilizadas na situação de vários processos estarem concorrendo a um único processador. Estas metodologias não são avaliadas na taxonomia de Casavant.

O **escalonamento global** envolve decidir o lugar (processador) no qual será executado determinado processo, ficando a tarefa de escalonamento local entregue ao sistema operacional do processador alocado. Esta clara separação objetiva minimizar a responsabilidade e o conseqüente *overhead* do mecanismo de escalonamento global. O desdobramento da taxonomia de Casavant, feito a seguir, diz respeito ao escalonamento global.

Escalonamento Estático & Dinâmico

Estas opções dizem respeito ao momento no qual as decisões sobre o escalonamento são tomadas.

Escalonamento Estático

Neste caso, as informações que definem quais são os processos da aplicação, bem como as que caracterizam as possíveis paralelizações de tarefas, estão

disponíveis no momento da geração do código executável. Os módulos objeto da aplicação são combinados com os módulos de carga, de tal forma que cada imagem executável gerada fica pré-destinada a um processador específico. Assim, as decisões são tomadas antes da execução ser iniciada e são fixas ao longo da mesma. Tipicamente, as informações que precisam estar disponíveis são o tempo estimado de execução dos processos a serem paralelizados, o volume de comunicação de cada um, as características dos processadores (sobretudo no caso de uma arquitetura heterogênea) e as latências de comunicação em função da topologia da rede de interconexões da arquitetura paralela.

Na taxonomia de Casavant, escalonamento estático é sinônimo de escalonamento determinístico.

Escalonamento Estático - Ótimo & Sub-ótimo

No caso onde todas as informações correspondentes ao estado do sistema e às necessidades de recurso dos processos são conhecidas, um escalonamento ótimo pode ser feito. Como exemplo de otimizações que podem ser atingidas, temos a minimização do tempo de execução, a maximização da utilização de recursos e a maximização do *throughput* do sistema. Muitas vezes a complexidade que atinge o escalonamento ótimo o torna computacionalmente intratável; daí surgem as soluções sub-ótimas.

Escalonamento Estático – Sub-ótimo – Aproximado & Heurístico

O escalonamento sub-ótimo **aproximado** utiliza os mesmos algoritmos do escalonamento ótimo, mas ao invés de explorar todo o espaço das possíveis soluções ideais, ele se satisfaz quando encontra uma considerada “boa”. Para as situações em que possam ser definidas métricas para reconhecer uma “boa” solução, esta alternativa pode reduzir consideravelmente o tempo necessário para obter um escalonamento para a aplicação em questão. A principal característica do escalonamento **heurístico** é o uso de parâmetros genéricos que sabidamente afetam o comportamento do sistema paralelo. Por princípio, estes parâmetros devem ser simples de obter ou calcular. Por exemplo, agrupar processos com elevada taxa de comunicação em um mesmo processador. A expectativa é que tal procedimento melhore o desempenho do sistema como um todo, porém não existe, neste caso, uma preocupação em garantir uma relação direta entre o mesmo e o resultado desejado. Em outras palavras, em função do conhecimento da dinâmica da arquitetura paralela, intuitivamente a utilização das heurísticas (e seus respectivos parâmetros) conduzirá a um melhor escalonamento, mas isto não pode ser antecipadamente provado.

Escalonamento Dinâmico

O escalonamento dinâmico tem por base que poucas informações a respeito das necessidades e do comportamento da aplicação estarão disponíveis de antemão. Deste modo, nenhuma decisão é tomada até que a mesma inicie sua execução. A decisão do lugar (em quais nodos processadores) os processos da aplicação serão computados é responsabilidade do ambiente de execução da arquitetura.

Escalonamento Dinâmico – Fisicamente distribuído & não distribuído

Neste nível, a taxonomia trata se o trabalho pertinente às tomadas de decisão de escalonamento estará distribuído entre processadores da arquitetura, ou se ficará a cargo de uma tarefa localizada em um único processador.

Escalonamento Dinâmico – Fisicamente distribuído – Cooperativo & Não cooperativo

O foco, neste nível do escalonamento distribuído, é o grau de autonomia que cada processador tem na determinação de como seus recursos podem ser utilizados. No caso cooperativo, cada processador tem a responsabilidade de contribuir com sua parte na tarefa de escalonamento, porém todos estão trabalhando segundo uma meta global da arquitetura. No modelo não-cooperativo, os processadores atuam como entidades autônomas e decidem o uso dos seus recursos sem considerar o efeito disto no resto da arquitetura. Para o escalonamento dinâmico distribuído e cooperativo se aplicam as mesmas sub-divisões existentes para o escalonamento estático, quais sejam: ótimo, sub-ótimo aproximado e sub-ótimo heurístico.

Classificação Horizontal

As características descritas na classificação horizontal de Casavant podem ser encaixadas em todos os ramos da sua classificação hierárquica. O objetivo da separação das classificações é tornar a taxonomia, como um todo, mais clara e objetiva. Não existe por parte do autor distinção de importância entre as mesmas.

Escalonamento Adaptativo & Não adaptativo

Uma solução **adaptativa** para o problema do escalonamento em sistemas paralelos e distribuídos, é aquela na qual os parâmetros, bem como os algoritmos utilizados para implementar a política de escalonamento, podem ser alterados, durante a execução, em função das respostas do sistema ao escalonador. Um caso típico, neste sentido, seria o escalonador em função do andamento da execução desconsiderar algum

parâmetro (ou reduzir a sua importância), se entender que o mesmo traduz uma informação inconsistente com relação às outras do sistema.

Em contraste, uma política **não adaptativa** não modifica seu mecanismo de controle de escalonamento em função do comportamento da execução paralela.

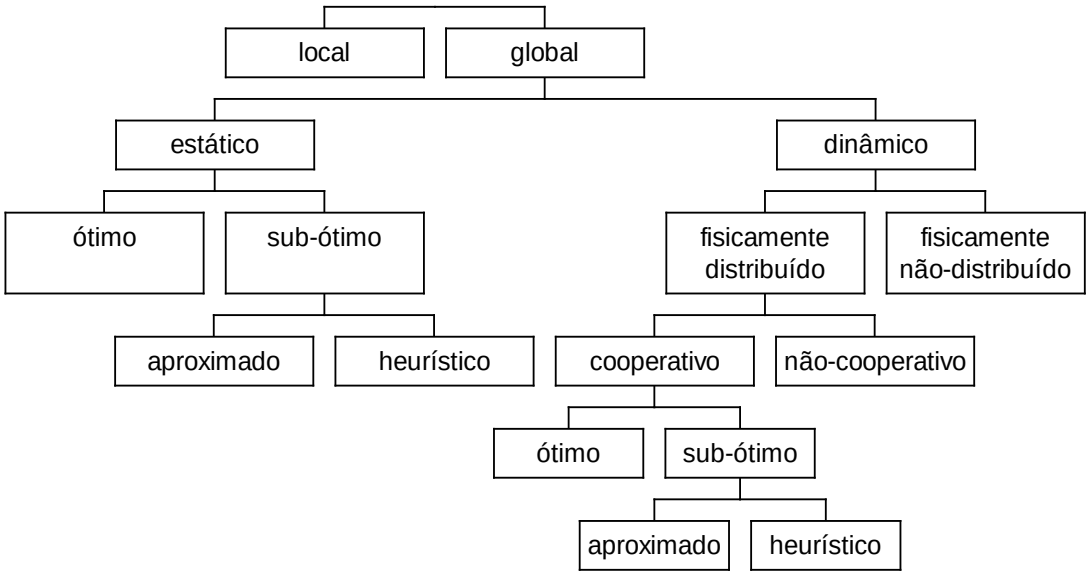

Figura 40: A Taxonomia de Casavant

Balanceamento de carga

A idéia básica desta política é fazer com que os processos progridam em todos os nodos a uma mesma razão. Para isto, a informação sobre a carga nos diferentes processadores é compartilhada através da rede de interconexão de forma periódica ou sob demanda, de forma que todos os nodos tenham uma visão do estado global do sistema. Assim, todos os nodos cooperam com o objetivo de remover trabalho dos processadores com elevada carga para os mais liberados. Esta estratégia é mais viável quando os nodos do sistema são homogêneos. Cuidados precisam ser tomados na escolha da unidade de medida da carga nos processadores, bem como para prevenir migrações de carga que resultem em baixa produtividade global.

Escalonamento por Licitação (Ofertas ou Leilão)

Nesta proposta de escalonamento, os nodos da arquitetura tanto podem assumir o papel de gerente, como de contratante. O nodo gerente é aquele que tem

tarefas para serem compartilhadas, e o contratante é o que tem disponibilidade computacional para executar novos processos. Os nodos gerentes anunciam a existência de tarefas a serem processadas e recebem propostas (ofertas de compra de tarefas) dos nodos contratantes.

O tipo e a quantidade de informações trocadas são determinantes para o desempenho desta proposta. Uma característica importante desta classe de escalonadores é o fato de que os gerentes têm autonomia para decidir, entre os nodos que responderam ao seu anuncio, para qual irá enviar a tarefa a ser executada (os critérios para decisão podem ser ótimos, sub-ótimos ou heurísticos). Some-se a isto que em função da evolução da computação na arquitetura como um todo, os contratantes não são obrigados a aceitar uma tarefa pela qual já manifestaram interesse.

Escalonamento Probabilístico

A motivação desta estratégia é a constatação que muitas vezes se mostra proibitivo o tempo necessário para tratar analiticamente todo espaço de soluções para o mapeamento de processos aos nodos da arquitetura. A idéia é gerar randomicamente (segundo alguma distribuição estatística) um mapeamento. Após terem sido gerados randomicamente diversos possíveis mapeamentos, o conjunto obtido é então analisado, e deste é escolhido o melhor mapeamento. Os critérios adotados são definidos em função do sistema paralelo como um todo (características de hardware e software).

Escalonamento de Atribuição Única & Reatribuição Dinâmica

Nesta classificação é considerado o momento em que o escalonador recebe as informações sobre a aplicação paralela a ser executada. Na proposta de Atribuição Única, as informações são disponibilizadas no momento que a aplicação é submetida à execução. Por sua vez, na Reatribuição Dinâmica, após uma execução parcial, são reavaliados os parâmetros passados inicialmente ao escalonador, utilizando informações dinamicamente criadas durante a própria execução. Este ciclo de execuções parciais e avaliação pode ocorrer diversas vezes durante todo processamento da aplicação.

A Reatribuição Dinâmica é um recurso que os administradores de sistemas paralelos têm para evitar que os usuários burlem a estrutura de prioridades anteriormente estabelecida, informando parâmetros que não correspondem à aplicação que será submetida à execução.

Escalonamento Estático e Dinâmico

Nesta seção serão analisados os escalonamentos estático e dinâmico: as suas características, as situações em que cada um tem o seu uso potencializado, bem como as principais estratégias e algoritmos empregados.

Escalonamento Estático

A proposta do escalonamento estático pode ser resumida em mapear um conjunto de tarefas parcialmente ordenadas nos processadores de uma arquitetura paralela, de tal forma que sejam reduzidos ao máximo os custos decorrentes de sincronização e/ou comunicação, e conseqüentemente o tempo para executar todas as tarefas (tempo total de execução) fique reduzido também.

As pesquisas nesta área mostraram que, exceto para casos bem restritos (tal como o escalonamento de tarefas com tempos de execução unitários e em modelos de arquitetura baseados em somente dois processadores [28]), a tarefa de encontrar o melhor escalonamento possível constitui um problema NP-completo. Em função disto, os trabalhos têm sido orientados no sentido de buscar soluções próximas à ideal, empregando para isto métodos aproximados ou heurísticos ([50]).

No escalonamento estático, o mapeamento das tarefas na arquitetura ocorre antes do início da execução. Uma vez iniciado o processamento, as tarefas permanecem nos processadores aos quais foram alocadas, até que a execução paralela seja encerrada, isto é, não serão passíveis de preempção ou de serem movidas a outro processador. Isto exige que o algoritmo de escalonamento tenha um bom conhecimento dos aspectos correspondentes à aplicação paralela a ser executada, tais como o tempo de execução das diversas tarefas, as relações de dependência entre as mesmas e os padrões de comunicação entre os processos. A decisão de escalonamento neste caso é centralizada, e não adaptativa.

Alternativas para o Escalonamento Estático

O escalonamento estático de aplicações paralelas tem sido trabalhado pela comunidade científica de forma intensa já há bastante tempo ([45]). Uma abordagem das alternativas até agora exploradas implica em extensos documentos ([50]). Neste item, será apresentada a natureza das diferentes propostas.

Dada uma aplicação paralela, constituída por diversos processos, e representada por um grafo caracterizando a relação entre os mesmos (*task graph*), e uma

rede de processadores também caracterizada por um grafo (*processors graph*), o assinalamento dos processos aos processadores, utilizando estes grafos com o objetivo de satisfazer determinadas métricas, constitui o escalonamento propriamente dito (é também conhecido como "problema de mapeamento ou alocação"). Especificamente nesta área, também existem trabalhos relevantes há algum tempo ([8]).

Existem duas alternativas para o grafo de processos de uma aplicação: o "grafo de precedência de processos" (*Task Precedence Graph* - TPG), também conhecido como *Direct Acyclic Graph* (DAG), e o "grafo de interação de processos" (*Task Interacting Graph* - TIG), o qual tem arcos sem assinalamento de sentido.

Um DAG representa aplicações paralelas, caracterizando as dependências de comunicação e/ou sincronização entre os processos que as formam.

Um TIG, por sua vez, é característico de aplicações paralelas cujos processos não apresentam uma ordem específica para execução (como quando somente existem barreiras de sincronização, situação típica de aplicações SPMD). O mapeamento dos processos na arquitetura pode levar em conta os custos de comunicação, bem como o poder computacional dos processadores envolvidos (quando se tratar de uma arquitetura não homogênea). Muitos autores não denominam este mapeamento como escalonamento, mas sim como "Alocação de Tarefas" (*task allocation*). Em [27] é trabalhada a prova de que a alocação de tarefas também é um problema NP-Completo. Este modelo é característico de um grande grupo de aplicações paralelas, tais como as utilizadas para resolver sistemas de equações em aplicações de elementos finitos ou simulação de sistemas de transmissão e distribuição de energia elétrica, aplicações para simulação de circuitos eletrônicos VLSI, etc.

Heurísticas para DAGs

As duas referências tomadas como base para esta síntese sobre escalonamento utilizando DAGs são [1] e [50]. Particularmente em [50], é feito um amplo levantamento sobre as características dos algoritmos mais relevantes, incluindo comparações e análises de desempenho dos mesmos.

Resumo dos Principais Tipos de Algoritmos para DAGs

BNP - *Bounded Number of Processors*: um algoritmo de escalonamento do tipo BNP escalona um DAG para um número limitado de processadores. Os processadores são assumidos como totalmente interconectados, e não existe nenhuma consideração quanto a possíveis contenções nos canais de comunicação.

UNC - *Unbounded Number of Clusters*: um algoritmo tipo UNC escalona um DAG a um número não limitado de grupos (clusters) de processadores. Os grupos gerados por estes algoritmos podem ser mapeados sobre os processadores utilizando um segundo algoritmo de alocação.
APN - *Arbitrary Processor Network*: um algoritmo tipo APN executa o escalonamento e o respectivo mapeamento dos processos, considerando a topologia da rede de interconexão de processadores utilizada na arquitetura. Neste tipo de algoritmo, é considerado de forma explícita o custo das comunicações sobre os canais entre os nodos. Deste modo, além de buscar minimizar o tempo de execução através de critérios de proximidade entre processos comunicantes, este tipo de algoritmo também tenta evitar que ocorram problemas de contenção durante a troca de mensagens entre os processadores.
TDB - *Task Duplication Based*: um algoritmo tipo TDB tem como ponto central a estratégia de duplicar processos com o objetivo de minimizar os custos (overhead) decorrentes das comunicações.

Escalonamento estático utilizando o modelo de precedência

O objetivo desta seção é, da forma mais resumida possível, caracterizar, através de exemplos, como ocorre o escalonamento estático a partir de um modelo de precedência entre as tarefas paralelizáveis. Inúmeras otimizações destes algoritmos básicos que serão apresentados nesta seção, têm sido propostas na literatura ([50], [27], [28]).

No modelo de precedência entre processos, o programa é representado por um DAG. Cada nodo no grafo representa uma tarefa com um tempo de execução conhecido. Um arco orientado representa uma relação de precedência entre duas tarefas; o mesmo é caracterizado por um valor (um peso) que especifica o tamanho da mensagem a ser transferida para a tarefa sucessora no momento que a tarefa origem for completada.

A figura 42a é um caso típico de DAG e caracteriza o grafo de precedência entre as tarefas da aplicação. A aplicação em questão consiste de 7 tarefas (de A a G), cada uma com o seu tempo de execução e com os respectivos volumes de comunicação definidos nos arcos.

Por sua vez, a figura 42b (grafo de comunicação do sistema) representa a arquitetura na qual as tarefas serão mapeadas. A mesma está caracterizada por um modelo de comunicação, o qual define a topologia de interconexão e os custos de comunicação entre os três processadores. Trata-se de um modelo sintético, bastante

utilizado, que registra os custos de comunicação sem contemplar os detalhes do hardware da arquitetura.

Cab = Custo interconexão P1-P2 x Volume de comunicação Ta-Tb

Onde:
o processador P1 aloja a tarefa Ta;
o processador P2 aloja a tarefa Tb.

Figura 41: Custo de Comunicação entre Duas Tarefas

De forma genérica, o custo de comunicação entre duas tarefas é computado multiplicando o custo de comunicação entre os processadores (que irão alojá-las), extraído do grafo de comunicação do sistema, pelo volume de comunicação entre as tarefas, obtido do grafo de precedência. A fórmula correspondente ao custo de comunicação entre as tarefas A e B está na figura 41.

Por exemplo, se as tarefas A e E na figura 42 foram escalonadas para os processadores P1 e P3, respectivamente, o custo de comunicação entre ambas será 8 (2 x 4).

Com base nos modelos da figura 42, serão analisadas três heurísticas básicas de escalonamento. Serão destacados, tão somente, os aspectos e os conceitos mais importantes pertinentes às mesmas. Suas descrições em detalhes podem ser encontradas em [28] e [17]. Os princípios destas heurísticas foram utilizados em diversas outras propostas mais otimizadas ([50]). Seus princípios também são utilizados em diversos algoritmos para escalonamento dinâmico.

Estratégia *List Scheduling* (LS)

Quando se utiliza o modelo de precedência entre as tarefas, o conceito de **caminho crítico** se mostra importante. O maior tempo entre os conjuntos de tarefas com execução obrigatoriamente seqüencial no DAG constitui o caminho crítico, e determina o limite teórico mínimo para o tempo total de execução.

Para o grafo de precedências entre tarefas da figura 42a, o caminho crítico tem o valor de 16 unidades de tempo (seqüências ADG ou AEG; a soma dos tempos de execução das tarefas em ambas é 16: 6 + 6 + 4).

O conceito de caminho crítico é freqüentemente utilizado na análise de desempenho de algoritmos heurísticos. Em muitos algoritmos este conceito é central na organização da sua estratégia de escalonamento das tarefas.

A estratégia *List Scheduling* é bastante simples, e constitui um exemplo típico de estratégia que não considera custos de comunicação. Nela, nenhum processador permanece livre, se existir alguma tarefa que possa ser executada no momento.

Uma heurística utilizada na mesma é mapear todas as tarefas do caminho crítico em um único processador. Por exemplo, na figura 43 as tarefas A, D e G estão mapeadas para o processador $P_{1.}$

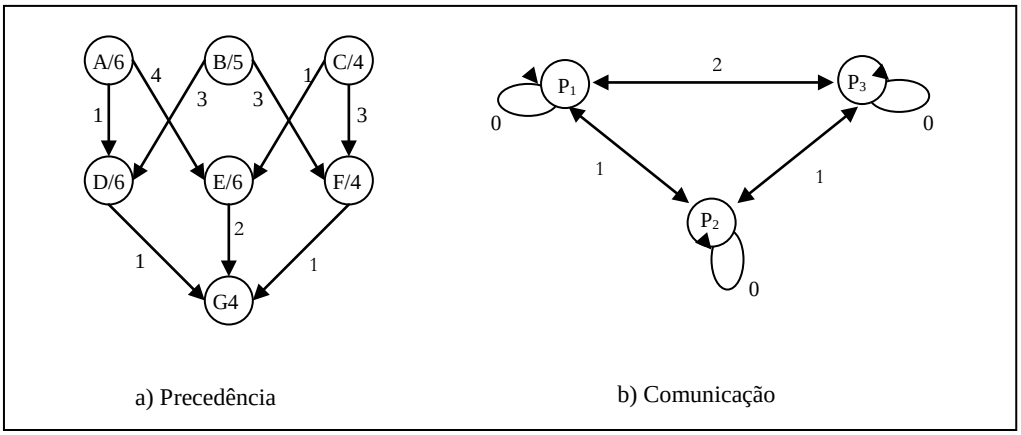

Figura 42: Modelos de Precedência entre Tarefas e de Comunicação do Sistema

Estratégia *Extended List Scheduling (ELS)*

Esta estratégia aloca as tarefas aos processadores utilizando *List Scheduling* como se o sistema fosse livre de custos de comunicação. Feito isto, acrescenta os custos de comunicação ao escalonamento obtido com LS. O custo de comunicação entre as tarefas é calculado de acordo com a fórmula apresentada na figura 41.

O escalonamento decorrente do uso da estratégia ELS para a aplicação e o sistema cujos grafos característicos estão na figura 42, resultou em um total para execução paralela de 28 unidades de tempo. Sua representação gráfica está na figura 43; as linhas pontilhadas na mesma representam a espera por comunicação (sincronização).

Esta estratégia básica muitas vezes não fornece bons resultados (como no caso em pauta). O problema central da mesma é decorrente das decisões de escalonamento serem feitas sem antecipar as comunicações.

Estratégia *Earliest Task First* (ETF)

Nesta estratégia, a tarefa que primeiramente estiver disponível será escalonada antes. O uso desta estratégia no mesmo exemplo fará com que a tarefa F seja escalonada após a tarefa E. Isto ocorre porque, se for considerado o custo de comunicações, a tarefa E estará disponível primeiro.

Conforme pode ser visto na figura 43, esta estratégia aplicada ao problema em questão gerou um tempo total de execução paralela de 18 unidades de tempo. Um resultado bem melhor que o obtido com a estratégia ELS.

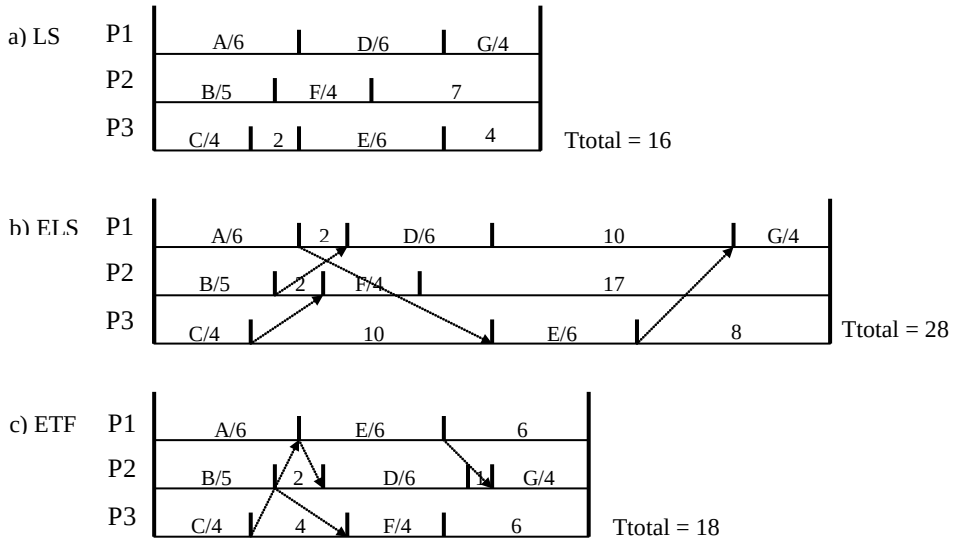

Figura 43: Escalonamentos Resultantes das Estratégias LS, ELS e ETF

Os modelos de precedência de processos e de comunicações no sistema são suficientemente genéricos para permitir modelar aplicações diversas, sobre uma arquitetura com custos de comunicação.

Mesmo este exemplo simples, cujo objetivo é caracterizar os aspectos mais usuais no escalonamento estático, mostrou que uma estratégia que produz ótimos escalonamentos para uma aplicação sobre um determinado sistema paralelo, não terá obrigatoriamente bons resultados sobre outro que tenha estrutura de comunicações diferente.

Este resumo de três estratégias básicas de escalonamento estático tem por objetivo exemplificar aspectos significativos ao tema. Por serem abordagens heurísticas, os resultados apresentados não traduzem superioridade de uma sobre a outra. Na abordagem heurística, os resultados são fortemente influenciados pelas características da aplicação e da arquitetura alvo do escalonamento.

Escalonamento Dinâmico

A idéia básica do escalonamento dinâmico é realizar o ordenamento e a alocação dos processos de uma aplicação paralela durante a execução da mesma. Seu uso é imprescindível quando a aplicação tiver um grau de não-determinismo que não permita o uso do escalonamento estático ([28]).

O escalonamento dinâmico introduz um custo extra ao sistema computacional durante o processamento da aplicação propriamente dita. Este custo é decorrente tanto da operação do próprio escalonador (consumo de processador para sua execução, as comunicações inerentes a sua atuação, etc.), como das ativações/movimentações de processos e/ou seus dados entre os nodos da arquitetura.

Nos últimos dez anos, diversos trabalhos foram propostos sobre o escalonamento dinâmico. Nas publicações [19], [56], [93], [65] e [85], podem ser encontrados estudos e comparações sobre os mesmos. Particularmente em [77], é apresentado um extenso levantamento sobre as alternativas para o escalonamento dinâmico.

Implementações do Escalonamento Dinâmico

A disponibilidade de processadores cada vez mais poderosos, a existência a nível comercial de redes de estações de trabalho, com velocidades nominais superiores a

100 Mbps, facultam configurações reais que atingem efetivamente o desempenho de supercomputadores. Porém, as redes de estações se diferenciam das arquiteturas paralelas tradicionais em função do caráter de imprevisibilidade que pode se revestir a ocupação dos processadores e dos canais de comunicação que a integram ([95]).

Deste modo, além do não-determinismo inerente a algumas aplicações paralelas, o uso cada vez maior de redes de estações de trabalho como arquitetura para processamento de aplicações paralelas e distribuídas, vem estimulando pesquisas na área de escalonamento e/ou balanceamento dinâmico de carga.

Em [44] é feito um estudo buscando compatibilizar as técnicas de escalonamento dinâmico empregadas em multiprocessadores convencionais, com os ambientes paralelos baseados em redes.

A seguir serão resumidas as estratégias básicas presentes nos diferentes algoritmos de escalonamento dinâmico. As referências apresentadas estão entre as primeiras correspondentes à respectiva estratégia. Desde estas primeiras publicações, diversos outros trabalhos abordando casos particulares de arquiteturas/aplicações, ou sugerindo otimizações, têm sido propostos ([77]). Por sua vez, na publicação [64], correspondente a um trabalho desenvolvido no II/UFRGS, pode ser encontrada uma ampla taxonomia sistematizando os conceitos pertinentes à área de escalonamento dinâmico.

Estratégia *Least-Loaded*.

Esta estratégia procura alocar as tarefas (processos) aos processadores menos carregados do sistema ([18]). Para reduzir custos de comunicação, alguns algoritmos que utilizam a estratégia *Least-Loaded* somente facultam fluxo de carga entre processadores de um mesmo grupo (*cluster*). Chega ao extremo, em alguns casos, dos algoritmos *Least-Loaded* somente encaminharem intercâmbio de tarefas entre processadores vizinhos.

O sistema UTOPIA e o LSF (*Load Sharing Facility*, versão comercial do UTOPIA) utilizam algoritmos de casamento de padrões para filtrar as estações inadequadas a determinada tarefa. Os critérios utilizados no casamento de padrões incluem tanto aspectos gerais, como o poder da UCP e tamanho da memória, até aspectos específicos e restritivos como a arquitetura do processador (em se tratando de ambientes com processadores heterogêneos). Os dois sistemas citados agrupam os processadores disponíveis em pequenos clusters, cada um com o seu algoritmo *Least-*

Loaded, para selecionar os processadores mais adequados para receber as novas tarefas (mais poderosos, mais desocupados) e que satisfaçam as exigências de recursos. Um aspecto interessante nestes sistemas é a existência de algoritmos que monitoram a atividade da rede de interconexão. Através desta monitoração, pode ser estimada a carga de trabalho dos processadores da arquitetura. Com isto, os sistemas UTOPIA e LSF conseguem reduzir significativamente o uso de mensagens específicas para atualizar informações de carga ([100]).

Estratégia *Threshold-Based*

Nesta estratégia, os processadores disparam ações de balanceamento de carga quando a sua carga excede um determinado limiar. Os algoritmos de escalonamento dinâmico *Threshold-Based* trabalham utilizando uma das três políticas a seguir([49]):

- política *sender-initiated*: os processadores mais carregados enviam seus processos para outros menos carregados;
- política *receiver-initiated*: os processadores que estão com cargas leves requisitam trabalhos dos sobrecarregados;
- política *symmetrically-initiated*: contempla uma combinação das outras duas políticas.

Como a estratégia *Threshold-Based* e suas variantes são bastante utilizadas, uma caracterização de seus procedimentos de trabalho será feita a seguir. Intencionalmente, a abordagem está construída de forma genérica e tem como referências [15] e [48].

Os Procedimentos da Estratégia *Threshold-Based*

Na política *sender-initiated*, a transferência de processos de um processador exportador para um importador exige três procedimentos básicos:

- Procedimento de Transferência: determina quando um nodo processador torna-se um exportador;
- Procedimento de Seleção: determina como o nodo exportador seleciona um processo para transferir;
- Procedimento de Localização: determina qual nodo pode ser o importador.

Se o tamanho da fila de processos (tarefas a executar) for indicador de carga, um nodo processador ativará o Procedimento de Transferência no momento em que receber um novo processo, e o tamanho da fila ultrapassar o limiar previsto. O processo recém chegado é o candidato natural para o Procedimento de Seleção, uma vez que para o mesmo o ônus de remoção será menor (poderão serão evitados custos característicos de preempção e sincronização do processo a ser exportado). O Procedimento de Localização, por sua vez, se reveste de uma complexidade maior, pois necessita que seja conhecido o estado global da arquitetura no tocante à distribuição das cargas nos nodos processadores. Quanto mais precisa a informação a respeito da distribuição de carga na arquitetura, mais eficiente será o Procedimento de Localização, porém maiores serão os custos que a mesma infligirá no ambiente de execução. Em um caso extremo, no qual nenhuma coleta de informação sobre a carga na arquitetura esteja prevista, o nodo importador pode ser escolhido de forma aleatória, porém neste caso é possível que ocorra um efeito dominó, caso o nodo importador escolhido já esteja sobrecarregado. Uma alternativa é o nodo exportador testar um determinado número de nodos (este limite é pré-fixado) e localizar o mais indicado para receber tarefa (o menos carregado). Mesmo antes de esgotar o intervalo limite de busca, esta pode ser interrompida, se for encontrado um nodo potencialmente adequado (baixa ocupação) para receber nova carga de trabalho. Naturalmente, se estiverem disponíveis na arquitetura recursos no hardware para comunicação de grupo (*broadcast*, *multicast*, etc.), os mesmos podem ser utilizados para otimizar o Procedimento de Seleção.

Os algoritmos da política *Sender-Initiated* têm um comportamento melhor em sistemas pouco carregados. Quando a carga global é baixa, o custo de encontrar um processador livre é menor, e os custos (*overhead*) de comunicação decorrentes da busca afetam menos o desempenho global do sistema.

Por sua vez, na política *Receiver-Initiated*, quando a carga do processador estiver abaixo de um determinado limiar, são ativados os algoritmos de busca de um nodo exportador. Para o Procedimento de Localização são utilizadas estratégias semelhantes à política *Sender-Initiated*. A decisão de qual processo remover do nodo exportador usualmente não é tão simples como na política *Sender-Initiated*, uma vez que este já pode estar em execução. Neste caso, os benefícios do compartilhamento de carga têm como contrapartida custos decorrentes de atividades de preempção e sincronização de processos, bem como os de comunicação inerentes a uma operação de migração (a qual pode implicar em transferência de todo contexto da execução).

Quando a carga global na arquitetura for elevada, a migração de processos na política *Receiver-Initiated* será pequena, e os eventuais processadores exportadores poderão ser encontrados com facilidade. Por sua vez, quando a carga no sistema for baixa, as eventuais comunicações (e a conseqüente ocupação da banda-passante da rede de interconexão) decorrentes da equalização da carga, exatamente por esta ser baixa, não afetarão significativamente o desempenho global do sistema.

Mostra-se oportuno combinar as duas políticas. Dependendo do estado da carga global do sistema, uma ou outra pode ser assumida para gerenciar o escalonamento dinâmico de tarefas. Neste caso, cada processador pode ser tanto exportador quanto importador. Em termos práticos, os nodos processadores implementam uma política de *rendevouz* e um serviço de registro pode ser utilizado para identificar os processadores mais carregados e os mais livres. Quando os limiares de importação/exportação são atingidos, a identificação do par importador/exportador pode ser feita através do mesmo. Uma versão desta estratégia foi implementada no escalonador de tarefas do ambiente de execução do projeto OPERA-E ([94]). Em termos gerais, as políticas *receiver-initiated* são mais estáveis que as *sender-initiated*.

Estratégia *Bidding*

Esta estratégia encara os processadores da arquitetura como recursos e os processos das aplicações como consumidores. Por exemplo, no sistema SPAW, que simula um mercado financeiro, são utilizados critérios de prioridade como dinheiro. As aplicações participam de "licitações" (concorrências) na busca de tempo de processamento, e somente as aplicações vencedoras executam seus processos nos nodos da arquitetura ([90]).

Particionamento em Arquiteturas Paralelas e Distribuídas

O compartilhamento de espaço (vide item 0 - Taxonomia de Feitelson) é feito dividindo os processadores da arquitetura paralela entre as aplicações em execução em determinado momento.

Em muitos casos, apenas o particionamento da arquitetura é tratado pelo sistema operacional. Nestes casos, o escalonamento propriamente dito é feito pelo ambiente de execução da aplicação, o qual realiza o mapeamento das tarefas da aplicação paralela nos processadores alocados (fisicamente reservados) para a mesma. Naturalmente, o escalonamento dentro de uma partição específica também pode ser

feito pelo sistema operacional da arquitetura. Em ambos os casos, ficam caracterizados escalonamentos em dois níveis.

De modo geral, é possível dizer que o particionamento é condicionado principalmente pelas características do hardware. Nem todas as arquiteturas paralelas apresentam a mesma facilidade para serem particionadas. Por exemplo, equipamentos SIMD ([38], [95]) não podem ser particionados, a menos que sua concepção seja alterada, de modo que fique possível associar uma unidade de interpretação de instruções e o respectivo mecanismo de propagação de instruções a cada partição alocada. Um exemplo desta limitação é a clássica CM-2; apesar de poder atingir 65536 processadores, a mesma pode chegar a ter no máximo 4 partições com possibilidade de executar aplicações de forma independente. Por opção do usuário, dois ou os quatro quadrantes podem ser combinados para executar um único programa.

Situações análogas ocorrem com outras arquiteturas

Os seguintes critérios estão entre os mais adotados para comparar os mecanismos de particionamento utilizados nas arquiteturas paralelas ([68]):

- **independência**: partições distintas devem ser tão independentes quanto possível. Deve ser evitado, sobretudo, o compartilhamento do hardware para chaveamento e/ou conectividade;
- **flexibilidade**: diz respeito à possibilidade de alocar partições de tamanho arbitrário, compostas por um subconjunto com qualquer número de processadores. De outra forma, será inevitável alguma perda por fragmentação;
- **dinamicidade**: as partições devem refletir as exigências das cargas de trabalho das aplicações paralelas que alojam. Usualmente, as cargas de trabalho oscilam durante todo período de execução, conseqüentemente deveriam variar também as dimensões das partições alocadas;
- **custo e complexidade**: estes aspectos devem ser baixos, tanto no tocante ao hardware necessário como no algoritmo exigido para o particionamento;
- **modularidade e escalabilidade**: as soluções para o particionamento devem ser válidas para os diversos tamanhos que possa vir a ter o sistema.

A Fragmentação em Arquiteturas Paralelas

A fragmentação em arquiteturas paralelas é um problema central a ser atacado pelas estratégias de particionamento. Sua presença implica em processadores não utilizados, o que compromete o desempenho e o índice geral de utilização do equipamento ([54]). A mesma pode ser classificada como interna ou externa. A **fragmentação interna** é conseqüência da alocação de tarefa paralela a uma partição maior que a necessária. Isto pode ocorrer em função do uso de partições de tamanho pré-definido, ou devido a restrições na estratégia de alocação (por exemplo, somente alocar partições proporcionais à potência-de-dois).

Por sua vez, a **fragmentação externa** ocorre quando a estratégia de alocação não consegue organizar os processadores disponíveis para atender as novas tarefas paralelas. Em função das suas causas, a mesma pode ser dividida em três tipos:

- fragmentação externa por insuficiência de recurso: a nova tarefa exige um número de processadores maior do que a arquitetura dispõe no momento. A nova tarefa não poderá ser alocada, e os processadores livres continuarão sem uso;
- fragmentação externa virtual: neste caso, a fragmentação é decorrente de um reconhecimento incorreto por parte do algoritmo de alocação da disponibilidade de processadores para atender a nova aplicação. O mesmo, por entender que a arquitetura não dispõe de processadores em número suficiente, não dispara a nova aplicação;
- fragmentação externa física: esta fragmentação é decorrente da dinâmica de uso da arquitetura. As diferentes aplicações, à medida que encerram, liberam processadores em diversas posições da arquitetura. Embora existam processadores livres em número suficiente ao exigido pela aplicação, a distribuição dos mesmos não permite que estes sejam utilizados para atender a nova aplicação. Este tipo de fragmentação é comum em ambientes com elevada dinâmica de execução (diversos usuários e/ou aplicações com diferentes naturezas), e constitui um dos principais fatores que concorrem para a sub utilização de equipamentos paralelos ([14]).

As Metas das Estratégias de Particionamento

As estratégias de particionamento, independentemente do método que utilizam, contemplam no mínimo três metas ([25]):

- minimizar a fragmentação: isto representa, dentre outros aspectos, potencializar a taxa de utilização;
- minimizar o diâmetro das partições: isto implica em reduzir os possíveis custos de comunicação;
- trabalhar com o menor custo computacional: isto significa buscar o melhor tempo de resposta para o algoritmo utilizado. Uma vez que o particionamento da arquitetura ocorre em tempo de execução, o tempo gasto com o particionamento, concorre na composição do tempo total que o usuário necessita aguardar até sua aplicação concluir.

Estas metas são conflitantes entre si. Isto explica a diversidade de propostas que têm surgido, todas buscando um ponto de equilíbrio entre as mesmas.

Por outro lado, a classificação das estratégias de particionamento se torna complexa pelo fato de três componentes envolvidos interagirem fortemente: a arquitetura, o sistema operacional e as aplicações. As propostas mais usuais classificam os mecanismos de particionamento em fixo, variável, adaptativo e dinâmico ([70]). Os mesmos serão analisados nas seções a seguir.

Particionamento Fixo

Nesta proposta, partições fixas são definidas pelo administrador do sistema paralelo. As características das partições são definidas em função do perfil das aplicações dos principais usuários (histórico de uso). Em muitas instalações paralelas existe uma reserva de processadores para as aplicações que interagem com o usuário, e o restante da arquitetura fica à disposição das aplicações que processam em *batch* ([35]).

Usualmente, os tamanhos das partições (e conseqüentemente o seu número) podem variar uma ou duas vezes durante o dia, para acomodar melhor as exigências da carga de trabalho. O perfil de trabalho durante o dia pode divergir significativamente do da noite. Uma discussão ampla sobre a carga de trabalho em arquiteturas paralelas pode

ser encontrada em [33]. Porém, no particionamento fixo, o tamanho das partições não é modificado para atender a demanda de aplicações específicas.

O maior inconveniente com o particionamento fixo é a fragmentação interna. As aplicações utilizarão ou não todos os processadores das partições para as quais serão alocadas. Para minimizar o número de processadores sem uso nas partições, uma solução é disponibilizar partições de diferentes tamanhos, de modo que o usuário possa escolher a que melhor se ajusta às suas necessidades. Contudo, mesmo que isto seja possível, o problema da fragmentação ainda persiste em menor escala, até mesmo porque muitas vezes as partições mais adequadas já poderão estar em uso.

Uma solução mais elaborada para o problema da fragmentação interna, é adotar um segundo nível de particionamento, no qual as partições fixas serão compartilhadas por *time-slicing* ou terão internamente particionamento variável (vide item 0). Este tipo de solução é adotado na Connection Machine CM-5, no IBM SP2, no Intel Paragon e no Meiko CS-2, dentre outros ([41], [20]).

É importante reiterar que, via-de-regra, quanto mais elaborada, maior será o custo que a estratégia de particionamento imporá ao sistema.

Particionamento Variável de Acordo com a Demanda

O particionamento variável difere do particionamento fixo pelo fato dos tamanhos das partições não estarem pré-definidos. Pelo contrário, são estabelecidos de acordo com as solicitações dos usuários. Isto é feito de duas maneiras: ou utilizando a combinação de tamanhos de partições pré-definidos, ou pela criação de partições com tamanhos completamente arbitrários.

A razão mais comum para utilizar a combinação de partições pré-definidas é a intenção de garantir um casamento do particionamento feito com a topologia da arquitetura do equipamento paralelo. Este é o caso, por exemplo, do particionamento de hipercubos. As aplicações para hipercubos são tipicamente projetadas considerando sua topologia, e em princípio não teria sentido executá-las em um conjunto arbitrário de nodos. Por isto, via-de-regra, os hipercubos são particionados em subcubos ([20]).

A fragmentação interna pode ocorrer quando o menor subcubo pré-definido tiver mais processadores do que aqueles que a aplicação exige.

Se a arquitetura não impuser restrições, partições arbitrárias podem ser criadas. Isto elimina a fragmentação interna, uma vez que não será alocada para a aplicação uma partição com mais processadores do que ela necessita. Contudo, o problema da fragmentação continua, agora de forma externa às partições alocadas. Isto ocorre porque o conjunto de processadores disponíveis pode não ter número ou topologia adequada, para atender as necessidades das novas aplicações paralelas.

Apesar das novas arquiteturas serem dotadas de redes de interconexão bastante flexíveis, é importante analisar o particionamento com o número de processadores baseado em potências-de-dois. Este tipo de particionamento foi muito utilizado, e ainda está presente em diversas arquiteturas em funcionamento.

Partições de Arquiteturas Baseadas em Potências-de-Dois

Em muitos casos, a estrutura do equipamento é baseada em potências-de-dois: os processadores são agrupados em pares, grupos de quatro, oito e assim por diante. Dois importantes exemplos, neste sentido, são os equipamentos paralelos baseados em topologia hipercúbica ou redes de conexão multiestágio. Em equipamentos multiestágio, cada partição contém um número em potência-de-dois de processadores e o mesmo número de módulos de memória. Nos hipercubos, cada partição é um hipercubo (subcubo) de menor dimensão. Ambas arquiteturas facultam partições independentes, mas sofrem de falta de flexibilidade, o que pode levar à fragmentação ([46], [20]).

Considerando as similaridades na proposta de construção destas duas arquiteturas, não é estranho que os mecanismos utilizados para o seu particionamento sejam praticamente idênticos.

A partir de uma grande partição livre, é simples atender a solicitação por uma partição menor, dividindo repetidamente a partição maior pela metade. O mais complexo é reagrupar as partições menores em uma maior à medida que as mesmas são liberadas, uma vez que o agrupamento precisa respeitar a topologia da arquitetura. Por exemplo, cubos 2-D somente podem ser combinados em um 3-D, se os mesmos forem adjacentes ([20], [47]).

O processo de alocação ganha complexidade quando ocorrerem problemas, ou com canais de comunicação, ou com o processador de algum dos nodos do subcubo a ser alocado. Esta situação de falha fica potencializada para toda arquitetura paralela com elevado número de nodos e o conseqüente número, ainda maior, de canais de

comunicação. Diversos trabalhos têm sido propostos nos últimos anos, contemplando técnicas (inclusive distribuídas) para detecção e localização de falhas; destacamos a recente publicação [62]. Naturalmente, dependendo da extensão da falha ou das atribuições dos nodos atingidos, a arquitetura como um todo poderá ficar inviabilizada.

Por sua vez, o procedimento de reunificação dos subcubos, à medida que as aplicações vão terminando, é bem mais complexo. O sistema precisa "reconhecer" se estes subcubos ociosos podem ser agrupados formando um subcubo de dimensão maior. Se eles não puderem ser agrupados diretamente, poderá ser vantajoso migrar alguma das tarefas paralelas em execução para outra região da arquitetura, e deste modo criar dois subcubos livres adjacentes que poderão então ser unificados. Em função disto, o reconhecimento de subcubos livres e seu potencial de reunificação tem recebido bastante atenção na literatura pertinente ([67], [17], [96]).

Como se pode inferir pelas datas, estes são os trabalhos que originaram as propostas. A partir dos mesmos, melhoramentos têm sido propostos. É necessário ter presente que a capacidade de reconhecimento de subcubos não é o único critério pelo qual estes algoritmos devem ser avaliados. É também importante que os subcubos possam ser reconhecidos rapidamente, isto é, que o algoritmo de reconhecimento não tenha uma complexidade computacional muito elevada.

Particionamento em Tamanhos Flexíveis

Nos equipamentos modernos, o impacto da topologia está diminuindo. Isto se deve aos avanços no hardware dos mecanismos de roteamento e comunicação. Atualmente, para algumas arquiteturas, já é possível trabalhar com a abstração de processadores totalmente interconectados e com uma distância uniforme entre os mesmos ([20]).

Esta situação faculta que partições de diferentes dimensões possam ser construídas, envolvendo processadores conectados a quaisquer portas da rede de interconexão.

Dentre outros aspectos, esta possibilidade é importante porque faculta que nodos defeituosos possam ser isolados sem invalidar todo um grupo (como pode ocorrer em arquiteturas hipercúbicas). Quanto maior o número de nodos na arquitetura, maior a probabilidade de algum apresentar defeito.

Esquemas de Alocação de Processadores no Particionamento em Tam. Flexíveis

A seguir, serão resumidas as características das estratégias mais conhecidas de alocação de processadores em partições de tamanho flexível. Estas técnicas têm como alvo arquiteturas com topologia em malha, característica bastante usual no mercado de arquiteturas paralelas (Intel Paragon, Parsytec, GC, Cray T3D, hpcLine Siemens, etc.).

Particionamento em Regiões Contíguas

Os esquemas de alocação de processadores relacionados a seguir buscam regiões contíguas para alocar partições. A expectativa é de que os custos com comunicação sejam minimizados. Somente mensagens da aplicação executando na partição são esperadas, e não existe o risco de contenção em algum processo da aplicação devido ao tráfego na rede de interconexão externa à partição. Por outro lado, esta condição reduz as chances de alocação imediata da tarefa, uma vez que o sistema poderá momentaneamente não dispor do número de processadores de forma contígua para atender a solicitação da tarefa paralela.

Os esquemas de alocação contígua mais conhecidos são: *Two Dimensional Buddy* (TDB) ([53]), *Frame Sliding* ([16]), *First-fit e Best-fit* ([101]), *Adaptative-Scan*, *Busy-List*, *Free-List*, *Limit* ([14]).

Particionamento em Regiões Não Contíguas

Considerando o já citado progresso nas redes de interconexão, a existência entre os nodos de um número maior de elementos de interconexão, nem sempre irá afetar significativamente a latência de comunicação entre os mesmos. Isto faculta que a alocação possa ser feita de forma não contígua.

Diversos algoritmos de alocação não contígua são analisados em [54]. As propostas mais difundidas são: *Random*, *Naive* e *Multiple Buddy System*.

A tabela 5 resume o comportamento da fragmentação em função do esquema de alocação empregado.

Tabela 5: Fragmentação Típica das Estratégias de Alocação de Processadores

Proposta	Fragmentação Interna	FE-Insuficiência de Recursos	FE-Virtual	FE-Física
TDB	Sim	Sim	Sim	Sim
Frame-Sliding	Não	Sim	Sim [a]	Sim
Fist-Fit e Best-Fit	Não	Sim	Alguma [a]	Sim
Adaptative-Scan	Não	Sim	Não	Sim
Busy-List	Não	Sim	Não	Sim
Free-List	Não	Sim	Não	Sim
Não contíguas	Não	Sim	Não	Não

[a] Somente acontece se a submalha disponível estiver em uma orientação contrária à solicitada

O estudo feito em [54] mostrou que alocar as partições de forma parcialmente não-contígua, isto é, de modo que exista algum grau de continuidade física entre grupos de processadores da partição, garante um desempenho melhor à aplicação que uma alocação de processadores totalmente esparsa. Isto se deve, sobretudo, à possibilidade de ocorrência de contenção no momento da troca de mensagens através da rede de interconexões. Em função da tecnologia atualmente disponível, a latência nas comunicações praticamente não se afeta com a distância entre os nodos, porém a possibilidade de ocorrência de contenção nos canais de comunicação mais concorridos nem sempre pode ser ignorada.

O custo computacional decorrente do uso de métodos mais qualificados (melhor capacidade de reconhecimento, alocação não-contígua, etc.) no particionamento de equipamentos de grande porte (elevado número de processadores), tem motivado trabalhos que propõem a utilização do paralelismo na computação dos algoritmos envolvidos no particionamento da arquitetura. Um exemplo neste sentido é o trabalho [23]. No mesmo, o particionamento é resolvido de forma cooperativa entre os nodos do próprio equipamento paralelo. A proposta é diluir a complexidade do algoritmo de alocação, utilizando de uma concepção paralela/distribuída do mesmo ([24], [25], [26]).

Particionamento Adaptativo

Os sistemas que, em função da carga atual da arquitetura, alocam um determinado número de processadores quando a aplicação é carregada, e que garantem este número durante toda execução, são denominados **adaptativos**. Por sua vez, as tarefas que se enquadram neste modelo são denominadas moldáveis (vide item 0). A

principal vantagem destes sistemas é a garantia de que o processamento da aplicação terá um ambiente isolado das flutuações de carga da arquitetura paralela.

O tamanho da Partição Definido Manualmente

Em termos práticos, a maioria dos sistemas paralelos têm o particionamento da arquitetura feito ou pelo administrador do sistema, ou pelo próprio usuário em um modelo primeiro-a-chegar-primeiro-atendido. É comum um misto de ambas propostas. O administrador do sistema eventualmente pode fazer uma reserva de processadores para alguma aplicação crítica; via de regra, enquanto existirem processadores, os usuários tem liberdade de alocá-los. Esta situação é comum em ambientes de pesquisa e ensino, e é utilizada no PC^2 (*Paderborn Center for Parallel Computing - http://www.upb.de/pc2/*) da Universidade de Paderborn, Alemanha.

No entanto, algumas estratégias para particionamento automático em função da demanda têm sido propostas e serão analisadas a seguir:

O Sistema Definindo o tamanho da Partição

Por não ser conhecida a demanda futura, o problema do particionamento *on-line* ganha complexidade. De forma genérica existem duas grandes possibilidades ([70]):

- reservar alguns processadores: isto faculta que alguns novos trabalhos possam ser aceitos para execução, porém como conseqüência alguns recursos do sistema poderão ficar sem uso, diminuindo a taxa geral de utilização;
- liberar a alocação de processadores: nesta proposta, se existir demanda, todos os processadores serão alocados. Como conseqüência, novas tarefas poderão ter de esperar para serem atendidas.
- buscando um compromisso entre minimizar o tempo de resposta das aplicações e maximizar o fluxo de carga (*throughput*) global da arquitetura, algumas políticas de particionamento automático foram propostas. É importante ter presente que estas duas métricas de desempenho, na maioria das situações, são conflitantes.
- uma metodologia para análise das principais políticas de alocação automática de processadores, para sistemas sem compartilhamento de

tempo, pode ser encontrada em [71].Os índices mais utilizados por estas políticas estão descritos a seguir:

***A* – Paralelismo médio de uma aplicação**: é o número médio de processadores utilizado pela aplicação, assumindo que um ilimitado número de processadores está disponível para a mesma;
***M, m* – Paralelismo máximo (*M*) e mínimo (*m*) de uma aplicação:** diz respeito ao número de processadores que a aplicação mantém simultaneamente ocupados; também é assumido que um ilimitado número de processadores está disponível para a mesma;
***PWS - Processor Working Set*:** o PWS de uma aplicação é o menor número de processadores que maximiza a eficácia da execução paralela, a qual é definida por:

$$E = \frac{S_p^2}{p}, \text{ onde } S_p \text{ é o } \textit{speedup} \text{ (redução no tempo de execução)}$$

$$S_p = \frac{T_{execução}^{1 \quad processador}}{T_{execução}^{p \quad processadores}}$$

quando p processadores são alocados para a aplicação.

As políticas de particionamento automático mais difundidas são :

A+: esta política é baseada no paralelismo médio da aplicação (*A*).
Run-to-Completion: esta política (RTC) necessita que o paralelismo máximo das aplicações seja conhecido;
***Processor Working Set*:** esta política visa atribuir a cada aplicação um número de processadores igual ao PWS das mesmas;
Threshold: nesta política o tamanho da partição é definido em função do número de aplicações que aguardam para serem executadas. Para cada valor (limiar), que o total de aplicações aguardando atinge, existe um respectivo tamanho de partições;
Robust Adaptive: a definição das partições é feita com base somente no estado atual da carga global da arquitetura paralela. O sistema está em um estado ideal quando o número de partições é o mesmo que o número de aplicações aguardando execução.

Maiores detalhes sobre as políticas de particionamento citadas podem ser encontradas em [81]. Neste trabalho, é feita uma detalhada análise das mesmas, a qual contempla inclusive avaliação de desempenho.

Particionamento Dinâmico

São denominados **dinâmicos** os sistemas que envolvem as aplicações em execução, nas flutuações da carga global de trabalho da arquitetura, e realizam modificações nos recursos já alocados para as mesmas (redução ou aumento no número

de processadores durante o processamento). As tarefas que suportam este comportamento são chamadas maleáveis.

A flexibilização do tamanho das partições faculta que as aplicações que serão submetidas ao sistema paralelo possam ser colocadas em execução imediatamente (sem aguardar em fila de espera), a menos que o número de aplicações, no momento, seja maior que o total de processadores da arquitetura.

O conceito de particionamento dinâmico também pode ser aplicado à execução de programas paralelos em redes de estações de trabalho. Nestas, as estações podem estar disponíveis (baixo uso) a qualquer momento, bem como podem ser solicitadas (elevado emprego dos seus processadores) pelos seus usuários a qualquer momento. É importante registrar que uma das primeiras implementações reais que efetivamente empregou particionamento dinâmico foi o projeto Piranha, o qual inicialmente foi implementado em redes de estações. O projeto Piranha foi portado para a arquitetura CM-5 ([12]).

Um importante pré-requisito para realocação dinâmica de processadores, é que não existam restrições para tal na topologia da arquitetura paralela. Deve ser possível agrupar virtualmente quaisquer processadores, sem incorrer em alterações significativas no desempenho das comunicações. Uma conseqüência do particionamento dinâmico é a possível interdependência entre as partições no tocante à rede de interconexão. Quando existir esta interdependência, os eventos de comunicação em uma partição podem afetar o desempenho das outras, o que é indesejado (vide critérios para análise dos mecanismos de particionamento no início da seção 0).

A Influência do Modelo de Programação

Muitas aplicações paralelas são escritas de modo que possam ser executadas em diferentes números de processadores. Porém, nem sempre este número pode ser modificado durante a execução. Este é o caso típico das aplicações que utilizam barreiras de sincronização e distribuem o paralelismo em função dos processadores disponíveis. Uma parada temporária de um processo, decorrente da remoção de um processador da partição previamente alocada, pode suspender a execução da aplicação como um todo ([15]).

Além da situação genérica da barreira de sincronização, a suposição que o número de processadores não pode variar durante a execução está muitas vezes inserida na própria proposta do ambiente de programação. Um exemplo neste sentido é o HPF

(*High Performance Fortran*); em tempo de compilação, o número de processadores não precisa ser conhecido, porém deve ser fixo durante a execução. Diversas funções intrínsecas da linguagem têm relação com o número de processadores. O número de processadores também é definidor na declaração e na distribuição das matrizes de dados ([86], [55] e [60]). Além do Fortran, ambiente muito utilizado em programação numérica, situação semelhante ocorre com outros ambientes baseados em troca de mensagens, dentre estes EUI ([3]), p4 ([10]), e PARMACS ([11]). Diversos estudos têm sido feitos no sentido de compatibilizar políticas de particionamento dinâmicas e o desenvolvimento de aplicações ([63]).

Uma das alternativas de programação que melhor se adapta à condição de particionamento dinâmico é o modelo *workpile*. Neste modelo, a aplicação é estruturada em múltiplos trabalhos independentes, os quais são localizados em uma fila. Os diversos processos da aplicação, executando em processadores distintos, retiram trabalhos da fila e realizam o correspondente processamento. Numa situação assim, novos trabalhadores podem ser incluídos a qualquer momento, bem como processadores podem ser retirados no momento que encerram a computação de uma tarefa. Tais alterações no número de processadores na partição alocada para a aplicação, afetam a ordem e a razão com que as tarefas são removidas da fila, porém não comprometem, como um todo, a consistência da execução paralela ([59], [66]).

É importante observar que a estruturação e a execução de aplicações segundo o modelo *workpile* é utilizado em muitas situações ([35], [33]). Porém, o emprego deste modelo não é condição indispensável para uso da técnica de particionamento dinâmico. Por exemplo, aplicações baseadas em troca de mensagens podem trabalhar em ambientes com particionamento dinâmico redistribuindo sua estrutura de dados. Como uma operação deste tipo, envolvendo movimentação de dados, implica em um custo adicional (*overhead*) significativo, é imperativo que o número de reconfigurações na prática permaneça baixo: caso contrário, os benefícios decorrentes de uma nova alocação de processadores serão cancelados pelos custos inerentes ao processo de reconfiguração. Alternativamente, podem ser utilizados modelos nos quais a reconfiguração das partições somente é facultada em determinados pontos. Estes pontos são escolhidos por caracterizarem os momentos da computação paralela nos quais a realocação de processadores tem menor custo, por exemplo, no início de um novo conjunto de laços paralelos ([85], [97]).

Políticas para Definição do Tamanho da Partição

A seguir serão resumidas as duas políticas mais difundidas ([65]).

Equipartitioning: esta política aloca de forma equânime os processadores da arquitetura entre todas as aplicações presentes. A cada chegada ou finalização da execução de aplicações, uma nova partição para todas as aplicações é calculada.

A mesma apresenta dois principais inconvenientes. Primeiro, uma aplicação pode ter sua execução temporariamente suspensa várias vezes (uma vez para cada aplicação que encerra, ou para cada nova aplicação submetida ao sistema), pois o custo total pode ser proibitivo. Segundo, atrasos podem ocorrer em função de sincronizações no momento do reparticionamento da arquitetura. Se ocorrer de uma aplicação encerrar imediatamente após o escalonador ter suspendido a execução das aplicações, o processamento na arquitetura ficará suspenso, até que os processadores da aplicação que encerrou sejam liberados (o que ainda é um custo menor que reativar as execuções e logo a seguir suspende-las).

Folding*:* esta política suspende a execução de pelo menos uma das aplicações em execução, a cada chegada ou finalização de aplicações no sistema. Deste modo, apresenta custos de sincronização no momento do particionamento bem menores que a política *Equipartitioning*. Se existirem processadores livres na arquitetura, o novo trabalho é iniciado utilizando todos estes. As aplicações em andamento não são interrompidas. Se não existirem processadores livres, o escalonador verifica o tamanho da maior partição atualmente alocada para uma aplicação. Se o tamanho for maior que 1, então a aplicação em execução na mesma é interrompida, e a metade dos processadores desta partição é alocada à nova aplicação .

Comparando as Duas Políticas de Particionamento Dinâmico

Segundo o trabalho [65], a política *Folding* consegue um melhor desempenho que a *Equipartitioning*. A vantagem a favor da política *Folding* aumenta à medida que cresce a razão com que chegam novas tarefas. A principal justificativa para isto e o elevado número de suspensões globais na execução paralela introduzidos pela política *Equipartitioning*.

Políticas de Escalonamento no Sistema Operacional

As políticas de escalonamento que serão tratadas nesta seção, via de regra, são combinadas com as estratégias de particionamento tratadas na seção 0. Deste modo,

em um mesmo equipamento, diferentes políticas de escalonamento podem estar em uso, atuando em partições diferentes. Será feita uma abordagem com foco nas principais estratégias, a partir das quais decorrem as variantes necessárias para atender os diversos casos específicos, tanto de arquiteturas (hardware) como de aplicações paralelas (software).

Processadores Independentes

A expressão "processadores independentes" significa que os processos das aplicações paralelas são administrados pelos nodos processadores de forma individual. Porém, ao nível da arquitetura como um todo, o conceito de agrupamento pode existir. Por exemplo, o sistema operacional pode trabalhar de forma independente a questão da criação de partições (alocação de processadores), da questão pertinente ao escalonamento de tarefas sobre a mesma. Neste caso, será criado um agrupamento de processadores (partição) que somente se envolve com os processos de uma determinada aplicação.

Dada a relação muitas-para-poucos entre tarefas e processadores, a questão é como mapear as tarefas nos processadores. Os processadores poderão trabalhar com as tarefas a processar utilizando em um extremo somente Filas Locais, ou todos os processadores poderão compartilhar em outro extremo uma única fila global (comum a todos). A escolha entre uma ou outra estratégia depende da natureza da arquitetura, mais especificamente da disponibilidade de recursos de memória compartilhada (ou de uma estratégia eficiente de DSM – *Distributed Shared Memory*). Estes aspectos serão discutidos no decorrer da seção.

Filas Locais

A utilização da estratégia de Filas Locais é a opção natural para arquiteturas de memória distribuída. Uma vez que somente um processador manipula a fila de processos a serem executados, não existe a possibilidade de contenção quando do acesso à mesma, nem se faz necessário o uso de estratégias de exclusão mútua (*locks*, semáforos, etc.).

Na estratégia de Filas Locais, cada processador atua como em um sistema monoprocessado, se fazendo necessário um outro mecanismo comprometido com o mapeamento (alocação) dos processos aos nodos da arquitetura paralela.

Alguns exemplos de sistemas de memória distribuída que utilizam a estratégia de Fila Local são: o sistema *PEACE*, que processa sobre a arquitetura Suprenum, o NX/2 sobre o iPSC/2, o SIMPLEX sobre o nCUBE e o próprio *Cosmic Cube* original, dentre outros ([74], [98], [75]).

A estratégia de Filas Locais também tem sido utilizada em alguns sistemas de memória compartilhada, a exemplo do *BBN Butterfly* (com os ambientes de execução *Crysalis* e *Psyche*), do Fujitsu AP1000 e do *MEMSY*, o qual tem os módulos de memória compartilhados por pares de processadores ([46], [42]).

Algumas arquiteturas dispõem de suporte de *hardware* para Filas Locais e trocas de contexto. Um exemplo típico é o Transputer, o qual foi peça central em muitos sistemas paralelos. Os Transputers foram projetados para trabalhar com o modelo de programação da linguagem Occam, o qual prevê o uso de múltiplas tarefas paralelas. A gerência destas múltiplas tarefas é feita por um eficiente mecanismo de troca de contexto implementado em *hardware*. Os sistemas operacionais para arquiteturas baseadas em Transputers fazem uso desta possibilidade e exploram multitarefa em cada nodo. Exemplos destes sistemas são o Trillium e o Hélios ([58]).

Como já comentado, tendo em vista a proposta de independência (baixo nível de sincronismo e/ou comunicações exigidos para o gerenciamento) dos processadores na estratégia Filas Locais, quanto mais eficiente for a alocação inicial de tarefas nos processadores da arquitetura, menor será a necessidade de políticas de balanceamento de carga, e maior poderá ser o desempenho global da execução.

Fila Global

Nesta estratégia, cada nodo da arquitetura comporta-se como em um sistema monoprocessador, executa uma cópia do sistema operacional e compartilha com todos os outros, algumas estruturas de dados. A principal estrutura comum a todos os processadores é a fila de controle de execuções de processos.

Os processos paralelizáveis das aplicações que estão prontos para serem executados são colocados nesta fila. Os processadores retiram o primeiro processo da fila, executam o mesmo por um período de tempo (um *quantum*), e retornam o mesmo para a fila. Esta proposta é especialmente comum em arquiteturas de memória compartilhada tipo UMA (*Uniform Memory Access*), tais como *Sequent Multiprocessor* e *SGI Multiprocessors Workstations*, e em sistemas operacionais como o Mach ([4], [6]).

O principal mérito da estratégia de Fila Global é que a mesma oferece um compartilhamento de carga automático.

Como se pode ver, na estratégia Fila Global, nenhum processador ficará desocupado se existir algum processo aguardando para ser executado. Isto, porém, se traduz em alguns custos:

- contenção no acesso à fila de processos: fato inerente à condição de recurso compartilhado. A contenção cresce com o número de processadores na arquitetura;
- perda da localidade de execução: os processos, via de regra, irão executar sobre diferentes processadores cada vez que forem escalonados. Como conseqüência, os processos não podem contar com a memória local para armazenar dados, e o seu estado na memória *cache* dos processadores será perdido a cada re-escalonamento. Esta situação pode ser contornada, em parte, utilizando critérios de afinidade no escalonamento. Neste caso, os processos serão re-escalonados no mesmo processador utilizado anteriormente ([87]).
- custos na sincronização: na proposta de Fila Global, não existe nenhum compromisso com a ordem na qual serão escalonados os processos das aplicações. Considerando que, em algumas aplicações paralelas, os processos podem ter elevada interação (situação de baixa granulosidade), a ausência de coordenação no escalonamento pode fazer com que ocorram situações nas quais os pares que precisam interagir não estejam ao mesmo tempo em execução. Neste caso, não poderá se realizar a citada interação, e poderão surgir trocas de contexto adicionais. A execução, como um todo, será penalizada com o *overhead* decorrente ([36]).

O escalonamento utilizando a estratégia de Fila Global, tem a propriedade de fazer com que o serviço recebido por determinada aplicação seja proporcional ao número de processos paralelizáveis que esta gera. Por um lado, esta propriedade é o caminho natural para que a aplicação que exige maiores recursos computacionais possa consegui-los; por outro, se mostra demasiadamente dependente do estilo (ou do ambiente) de programação do usuário. Mecanismos para manter sobre controle este aspecto têm sido estudados ([34]).

Combinação de Filas Locais e Fila Global

A tabela 6 resume as principais diferenças entre as estratégias de Filas Locais e Fila Global. Como pode ser visto, cada estratégia tem as suas vantagens e desvantagens. Alguns trabalhos foram feitos no sentido de combinar as boas características de ambas.

Um sistema operacional que utiliza esta combinação é o MAXI utilizado no *Makbilan Research Multiprocessor*. Neste sistema, uma fila global é utilizada na distribuição do trabalho, e as filas locais no gerenciamento dos processos já alocados. O mapeamento dos processos é feito pelo sistema operacional através de um serviço específico (`get_work`). Os processadores menos carregados da arquitetura têm preferência no recebimento de serviço. Uma vez mapeados pelo `get_work`, os processos não migram ([72]). Estratégia semelhante é utilizada no KSR1 (*Kendall Square Reserarch*).

Um outro sistema que explora a combinação de filas locais e fila global, é a variante do sistema operacional Mach desenvolvido para o IBM RP3.

Tabela 6: Comparação Entre as Estratégias de Filas Locais e Fila Global

Característica	Filas Locais	Fila Global
Aplicabilidade	memória distribuída	memória compartilhada
Distribuição da carga	exige balanceamento de carga explícito	compartilhamento de carga automático
Localidade no contexto da aplicação	Mantém	no melhor caso parcial, se forem utilizados critérios de afinidade no escalonamento
Custo operacional (*overhead*)	baixo, sem contenção	exige o uso de recursos de exclusão mútua (*locks*), e pode sofrer contenções

O escalonamento no Mach do RP3 é baseado no conceito de famílias de processos. Cada família tem a sua fila global, e os processadores alocados a determinada família executam os processos associados a esta fila. Além da terminologia, a diferença principal em relação ao mecanismo original de alocação de processadores utilizado no CMU Mach é a opção de mapear certos processos a processadores específicos. No Mach RP3, a cada troca de contexto, os processadores podem escolher o próximo processo a ser executado, ou da sua fila local, ou da fila

global, dependendo das respectivas prioridades. Esta característica permite compartilhamento de carga a partir da fila global, associada com a vantagem de baixa contenção de uma fila local. Some-se a isto a possibilidade de dedicar poder computacional a processos prioritários, associando os mesmos a processadores específicos ([9]).

Também no sentido de reduzir a contenção inerente ao uso de uma fila global, em [21] e [22] foi proposto um sistema hierárquico de filas. Nesta proposta, toda nova tarefa é assinalada na fila global de tarefas do sistema. Entre esta fila global e a fila local em cada processador, existe uma hierarquia. Quando a fila local do processador estiver vazia, ele tenta buscar mais tarefas do seu "pai" (processador cuja fila é imediatamente superior na hierarquia); caso a fila do "pai" não tenha tarefas para compartilhar, a requisição de tarefas é propagada em direção ao topo da hierarquia.

Escalonamento de Grupos – *Gang Scheduling*

Em sistemas paralelos multiprogramados (com *time-slicing*), coexistem processos provenientes das diversas aplicações paralelas que compartilham a arquitetura. Os processos de uma mesma aplicação formam um grupo e cooperam entre si (interagem), e disputam com os outros grupos os recursos da arquitetura. Diversos trabalhos acadêmicos, bem como vários fabricantes de arquiteturas paralelas, acreditam ser esta uma alternativa promissora ([79], [78], [51], [43], [30], [4]).

Definição e Motivação

É entendida como *Gang Scheduling* a estratégia de escalonamento que combina as três características a seguir ([34]):

- os processos das aplicações são associados a grupos (*gangs*);
- os processos de cada grupo executam simultaneamente em processadores distintos, utilizando um mapeamento de um-para-um;
- é utilizado compartilhamento de tempo (*time-slicing*); com isto, todos os processos de um grupo têm sua execução suspendida (sofrem preempção) e são re-escalonados ao mesmo tempo.

O uso da estratégia *Gang Scheduling* confere algumas características ao sistema paralelo:

Abstração de um Equipamento Dedicado: *Gang Scheduling* faculta a abstração de um equipamento dedicado para todas as aplicações que compartilham a arquitetura, e não impõe restrições quanto ao modelo de programação a ser utilizado. Esta abstração de um equipamento dedicado tem diversas repercussões; por exemplo, como todos os processos da aplicação estão em execução simultaneamente, é possível que os mesmos trabalhem com interações de baixa granulosidade. Também neste sentido, é possível trabalhar com barreiras de sincronização, sem o risco do conjunto ficar aguardando um processo que não está em execução ([73]). No tocante às comunicações, mensagens assíncronas podem ser utilizadas sem o risco de que seja excedida a capacidade dos *buffers* do sistema. Os recursos de hardware para comunicação, por sua vez, podem ser acessados diretamente pelos próprios usuários, sem a necessidade de sofisticados mecanismos de comunicação. A estratégia de mapeamento um-para-um permite, sobretudo, que os processos sejam associados com estruturas de dados nas memórias locais dos nodos processadores (localidade no acesso às estruturas de dados - [29]).

Maior Interatividade com os Usuários: é importante ter presente que outras estratégias de escalonamento têm esta mesma meta (por exemplo, a estratégia de Particionamento Dinâmico - vide seção 0). A vantagem do *Gang Scheduling* é que, em função do uso de *Time-Slicing*, o mesmo consegue evitar que aplicações de grande porte monopolizem a arquitetura por longos períodos de tempo. Como resultado, é oferecida uma maior interatividade aos usuários que submetem pequenas tarefas no sistema. Por outro lado, o uso de preempção introduz diversos custos adicionais (*overheads*) e pode reduzir a eficiência do sistema de memória cache como um todo ([37]).

Compartilhamento: uma terceira razão para uso do *Gang Scheduling*, é que o mesmo faculta compartilhamento em arquiteturas que não podem ser particionadas. Por exemplo, é impossível utilizar políticas de particionamento flexível em uma arquitetura SIMD (em função da unidade de controle centralizada e do mecanismo de *broadcast* de instruções). O uso de *time-slicing* permite que diversas aplicações paralelas concorram simultaneamente à arquitetura. Esta alternativa foi utilizada na CM-2 para potencializar o acesso de usuários ([37]).

Como são Estabelecidos os Grupos (*gangs*)

Na maioria dos casos, todos os processos paralelizáveis da aplicação formam um único grupo. Esta visão, por exemplo, é totalmente compatível com o estilo SPMD (*Simple Program Multiple Data*) de programação, no qual é intrínseco que todos

os processos da aplicação distribuídos na arquitetura executem simultaneamente, isto é, ou todos estão em execução, ou nenhum está ativo.

Para que seja viável trabalhar com a situação na qual o número de processos da aplicação é maior que o número de nodos processadores da arquitetura, existem critérios para realizar o necessário agrupamento. Um critério básico é formar um grupo com os processos que interagem em baixa granulosidade. A definição dos integrantes do grupo pode ser feita observando a interação entre os processos durante parte da execução ([32]).

Um outro critério é utilizar estruturas sintáticas na programação para definir os grupos. Exemplos de estruturas usualmente empregadas são `parfor` ou `parbegin`, `par` ou `spaw.` Neste caso, quando os processos são distribuídos juntos por um mesmo construtor paralelo, pode ser assumido que devem constituir um grupo ([84]).

Lazy Gang Scheduling

Na sua proposta básica, o *Gang Scheduling* trabalha com um valor de tempo pré-definido e comum para todas as aplicações, durante o qual seus processos permanecem em execução até sofrerem preempção segundo um critério *round-robin*.

Na proposta *Lazy*, cada aplicação tem um tempo máximo de espera na fila de execução. O tempo de espera é baseado na sua classe. Por exemplo, aplicações interativas e de depuração, recebem tempos de espera menores que aplicações executadas na forma *batch* (execução de forma automatizada sem a participação do usuário). No momento em que o tempo de espera de uma aplicação é excedido, sua prioridade cresce, e a aplicação de menor prioridade em execução no sistema tem sua execução suspensa para liberar espaço para a mesma.

A aplicação escalonada tem então um certo tempo de execução garantido (*do-not-disturb time*), o qual é proporcional (dentro de uma faixa) a parâmetros da aplicação (por exemplo, a ocupação de memória). Transcorrido este tempo, a aplicação é candidata à preempção se existir outra de maior prioridade aguardando para ser executada na fila de espera.

A proposta de *Lazy Gang Scheduling* foi desenvolvida no *Lawrence Livermore National Lab* e está disponível para arquitetura Cray T3D ([37]). Uma

variante desta proposta utilizando informações decorrentes da própria execução (*feedback*), foi apresentada em [76].

Implicações no Desempenho

Em função do uso de compartilhamento de tempo (*time-slicing*), a proposta *Gang Scheduling* pode atender tantas aplicações quantas as filas de controle do sistema possam administrar.

Esta característica de pronto-atendimento ao usuário, contudo, dificulta a previsão do desempenho das aplicações, uma vez que não existe uma previsão da carga global à qual vai ser submetida a arquitetura paralela. Por outro lado, é desejada em muitos casos a garantia de uma razão na evolução do processamento de determinadas aplicações. As propostas no sentido de garantir um desempenho mínimo para certas aplicações, empregam uma estratégia de "reserva de tempo" a cada ciclo de escalonamento.

Esta idéia é utilizada no escalonador *Vanilla Gang Scheduling* ([91]). Na filosofia do mesmo, as aplicações são classificadas em função do seu grau de paralelismo (cujo valor é ajustado para a potência-de-dois superior mais próxima), esta informação é utilizada para particionar a arquitetura em função da classe de aplicações que vai ser executada. O escalonador aloca uma determinada fração de tempo para cada classe de aplicações, independentemente de quantas aplicações estão associadas à mesma. Como o administrador tem controle na criação de classes e na alocação de aplicações às mesmas, é possível priorizar a execução de determinadas tarefas em detrimento de outras.

Tendências na Área de Escalonamento

Para uma área tão dinâmica como a do escalonamento, uma atitude necessária é manter uma visão questionadora sobre as novas direções que se apresentam, tendo em vista as tendências do hardware e do software para suporte à execução paralela e distribuída.

Inúmeras publicações têm sido feitas na área de escalonamento, nos últimos anos. A despeito deste grande volume de trabalho já aportado, muito ainda está para ser feito. Diversas frentes de pesquisa estão em destaque atualmente. Dentre estas, temos:

- **integração do escalonamento com outros serviços do sistema operacional**: destaca-se neste caso a integração com o gerenciamento de memória ([5]);
- **caracterização das cargas de trabalho** (para instalações em regime de produção): os diversos pesquisadores que trabalharam com o tema, apesar das especificidades de seus trabalhos, chegaram ao ponto comum que os usuários via-de-regra solicitam mais recursos (processadores, etc.) do que aqueles que efetivamente necessitariam. Deste modo, com o uso de técnicas específicas para caracterização das diferentes cargas de trabalho (baseadas, por exemplo, em registro e análise de *logs*), seria possível, mesmo sem uma cooperação maior de parte dos usuários, otimizar os recursos destinados a cada aplicação ([82]);
- **utilização futura otimizada**: este problema é típico de sistemas onde aplicações do tipo moldáveis (vide item 0) são ativadas utilizando uma política de escalonamento baseada em uma fila FCFS (*First-Come-First-Served*). A questão é decidir entre iniciar o quanto antes a aplicação com os processadores livres, ou postergar sua execução até que um número maior de processadores esteja disponível. Por outro lado, a investigação de estratégias que apontem o total de processadores que deverão ser deixados livres em antecipação a futuras chegadas de tarefas, também é objeto de estudos ([83]);
- **emprego de algoritmos genéticos**: no caso mais genérico, o escalonamento de tarefas em arquiteturas paralelas, como já discutido, é um problema NP-Completo. Isto leva à utilização de heurísticas, as quais nem sempre atingem resultados igualmente bons para os diferentes espaços de solução do problema. Uma alternativa para superar esta condição é a utilização de uma heurística que possa se adaptar às especificidades dos problemas. Uma possibilidade para obter esta heurística auto-adaptável é a utilização de algoritmos genéticos ([103]).
- **utilização de compartilhamento de tempo**: apesar dos custos inerentes à preempção de processos, as alternativas decorrentes do seu uso permitem a construção de escalonadores mais flexíveis. É possível observar uma presença constante de publicações envolvendo *Gang Scheduling* nos principais eventos internacionais ([80], [99], [79]);

- **escalonamento aplicativo**: o objetivo do trabalhos [39], [69] e [7] é dissociar o balanceamento de carga tanto do programa em execução quanto da arquitetura destino. Com isto é melhorada a portabilidade, pois não é necessário considerar na programação as características do hardware paralelo/distribuído. Por outro lado, como o balanceamento de carga é dissociado do código aplicação, este pode se valer de diferentes algoritmos, e deste modo utilizar o mais adequado ao par aplicação-arquitetura. Para isto, nos trabalhos citados, a interface do ambiente de execução com a aplicação é representada por um grafo. Particularmente na linguagem Athapascan-1 [39], este grafo é construído em tempo de execução. O mesmo descreve as tarefas criadas e as respectivas comunicações entre estas, a manipulação deste grafo pelo núcleo de escalonamento viabiliza o balanceamento de carga.

- **utilização de paralelismo no escalonamento e no particionamento**: o crescimento do número de nodos nas arquiteturas paralelas, tem estimulado a pesquisa de ambientes de execução que empreguem a paralelização ou a distribuição no escalonamento/particionamento. Um exemplo da tendência das arquiteturas modernas, é o equipamento ASCI White da IBM. O ASCI White é capaz de atingir níveis de processamento da ordem de 12,3 *Teraflops*, sendo formado por 8.192 processadores organizados na forma de cluster hierárquico (linha RS/6000 SP).

Além dos acima descritos, diversos outros tópicos de pesquisa têm se mostrado presentes na comunidade científica: métricas e *benchmarkings* no escalonamento de sistemas paralelos e distribuídos, modelos de cargas sintéticas para avaliar escalonadores, técnicas dinâmicas de particionamento de arquiteturas e as aplicações maleáveis, escalonamento utilizando duplicação de processos, escalonamento em processadores heterogêneos, dentre outros.

Na tentativa de resumir a pesquisa na área de escalonamento de sistemas paralelos e distribuídos, é possível dizer que a mesma busca propiciar, à comunidade usuária, técnicas que, integradas aos modelos de programação, ofereçam para as alternativas de hardware existentes, um ambiente que: maximize a *utilização* e o *throughput* do equipamento, reduza o tempo médio de resposta das aplicações, e garanta a cada um dos usuários simultâneos a comodidade de uma máquina dedicada. Metas estas em muitas circunstâncias contraditórias, e que precisam ser perseguidas em um

mercado cujos patamares tecnológicos, tanto do hardware como do software, estão em constante transformação.

Referências

1. AHMAD, Ishfaq; KWOK, Yu-Kwong. On Parallelizing the Multiprocessor Scheduling Problem. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.10, n.4, p.414-432. April 1999.
2. ANDREWS, Gregory R**. Concurrent Programming: Principles and Practice**. Redwood City. The Benjamin/Cummings. 1991. 637p.
3. BALA, V. et al. The IBM External User Interface for Scalable Parallel Systems. **Parallel Computing**. v.20, n.4, p.445-462. Apr 1994.
4. BARTON, J.; BITAR, N. A Scalable Multi-discipline, Multiple-processor Scheduling Framework for IRIX. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.182-199 (Lecture Notes in Computer Science, v.949).
5. BATAT, Anat; FEITELSON, Dror. Gang Scheduling with Memory Considerations. **Proceedings…** of the 14th International Parallel & Distributed Processing Symposium – IPDPS 2000, Cancun, Mexico, IEEE, 2000. 842p. p.109-114.
6. BLACK, D. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. **Computer**. New York, v.23, n.5, p.35-43. May 1990.
7. BLUMOFE, R.D. et al. Cilk: an Efficient Multithreaded Runtime System. ACM SIGPLAN Notices. v.30. n.8, p.207-216. Aug 1995.
8. BOKHARI, S. H. On The Mapping Problem. **IEEE Transactions on Computers**. New York, v.C-30, p.207-214. Mar 1981.
9. BRYANT, R. M.; CHANG, H-Y.; ROSENBURG, B. S. Operating System Support for Parallel Programming on RP3. **IBM Journal of Research and Development**. New York, v.35, n.5-6, p.617-634. Sept/Nov 1991.
10. BUTLER, R.; LUSK, E. Monitors, Messages, and Clusters: the p4 Parallel Programming System. **Parallel Computing**. v.20, n.4, p.547-564. Apr 1994.
11. CALKIN, R. et al. Portable Porgramming With the PARMACS Message Passing Library. **Parallel Computing**. v.20, n.4, p.615-632. Apr 1994.
12. CARRIERO, N.; FREEMAN, E; GELERNTER, D. Adaptative Parallelism on Multiprocessors: preliminary experience with Piraña on the CM-5. In: Languages and Compilers for Parallel Computing, 1993. **Proceedings…**Berlin. Spring-Verlag, 1993. p139-151. (Lecture Notes in Computer Science, v768).
13. CASAVANT, Thomas L.; KUHL, Jon G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Transactions on Software Engineering**, New York, v. 14, n. 2, p.141-154, Feb. 1988.
14. CHANG, Chung-yen; MAHAPATRA, Prasant. Performance Improvement of Allocation Schemes for Mesh-Connected Computers. **Journal of Parallel and Distributed Computing**. New York, v.52, n.1, p.40-67, July 1998.
15. CHOW, Randy; JOHNSON, Theodore. **Distributed Operating Systems and Algorithms**. Berkeley: Addison Wesley Longman, Inc, 1997. 569p.
16. CHUANG, P. J.; TZENG, N. F. Allocating Precise Submesh in Mesh-Connected Systems. **IEEE Transactions on Parallel and Distributed Systems**. New York, p.211-217. Feb 1994.
17. CHUANG, P.; TZENG, N. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. **IEEE Transactions on Computers**. New York, v.41, n.4, p.467-479. Apr 1993.

18. CLARK, H.; McMILLIN, B. DAWGS – A Distributed Compute Server Utilizing Idle Workstations. **Journal of Parallel and Distributed Computing**. New York, v.14, n.2, p.175-186. Feb 1992.
19. CORRADI, Antonio; LETIZIA, Leonardi; ZAMBONELLI, Franco. Diffuse Load-Balancing Policies for Dynamic Applications. **IEEE Concurrency**. New York, v7, n.1, p.22-31. Jan-Mar 1999.
20. CULLER, D. E. and SINGH, J.P. Parallel Computer Architecture: a hardware and software approach. Morgan Kaufmann. 1999. 479p.
21. DANDAMUDI, S. P. Reducing Run Queue Contention in Shared Memory Multiprocessors. **Computer**. New York, v.30, n.3, p.82-89. Mar 1997.
22. DANDAMUDI, S.P.; CHENG, P.S.P. A Hierarquical Task Queue Organization for Shared-Memory Multiprocessor Systems. **IEEE Transactions on Parallel and Distributed Systems**. New York, v.6, n.1, p.1-16. Jan 1995.
23. DE ROSE, César A. F.; NAVAUX, Philippe O. A.; Geyer, Claudio R. Distributed Processor Allocation in Mesh-Connected Multicomputers. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APLICATIONS (PDPTA'2000), Jun. 2000, Las Vegas, USA.
24. DE ROSE, Cesar. **Distributed Processor Management in Multicomputers**. Karlsruhe. University of Karlsruhe. 1998. (Phd Thesis).
25. DE ROSE, Cesar; NAVAUX, Philippe. A Distributed Model for Processor Allocation and Management in Multicomputers. **Proceedings…** of the 9th Brazilian Symposium on Computer Architecture an High Performance Computing – SBAC-PAD97, Campos do Jordão, Brazil, 1997.
26. DE ROSE, Cesar; NAVAUX, Philippe. Algoritmos Paralelos para Gerência e Alocação de Processadores em Máquinas Multiprocessadoras Hipercúbicas. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 5., Set. 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. 775p. p.459-474.
27. EL-REWINI et al. **Task scheduling in parallel and distributed systems**. Englewood Cliffs: Prentice-Hall, 1994. 290p.
28. EL-REWINI, Hesham; LEWIS, Ted. **Distributed and Parallel Computing**. Greenwich: Manning Publications Co, 1998. 447p.
29. FEITELSON D. G.; RUDOLPH, L. *Gang Scheduling* Performance Benefits for Fine-Grain Synchronization. **Journal of Parallel and Distributed Computing**. New York, v.16, n.4, p.306-318. Dec. 1992.
30. FEITELSON, D. G. Packing Schemes for *Gang Scheduling*. In: Job Scheduling Strategies for Parallel Processing, 1996. **Proceedings…** Berlin. Spring-Verlag, 1996. p.1-26. (Lecture Notes in Computer Science, v.1162).
31. FEITELSON, D.G.; RUDOLPH, L. Distributed Hierarchical Control for Parallel Processing. **Computer**. New York, v.23, n.5, p.65-77. May 1990.
32. FEITELSON, Dror G., RUDOLPH, Larry. Coscheduling Based on Runtime Indentification of Activity Working Sets. **Journal of Parallel Programming**. New York, v.23, n.2, p.135-160. Apr 1995.
33. FEITELSON, Dror G., RUDOLPH, Larry. Metrics and Benchmarking for Parallel Job Scheduling. In: Job Scheduling Strategies for Parallel Processing, 1998. **Proceedings…** Berlin. Spring-Verlag, 1998. p.1-24. (Lecture Notes in Computer Science, v.1459).
34. FEITELSON, Dror G., RUDOLPH, Larry. Parallel Job Scheduling: Issues and Approaches. In: 1st. Workshop on Job Scheduling Strategies for Parallel Processing, 1995. **Proceedings…** Berlin. Spring-Verlag, 1995. (Lecture Notes in Computer Science, v.949).
35. FEITELSON, Dror G., RUDOLPH, Larry. Toward Convergence in Job Schedulers for Parallel Supercomputers. In: Job Scheduling Strategies for Parallel Processing, 1996. **Proceedings…** Berlin. Spring-Verlag, 1996. p.1-26. (Lecture Notes in Computer Science, v.1162).
36. FEITELSON, Dror; et al. Theory and Pratice in Parallel Job Scheduling. In: Job Scheduling Strategies for Parallel Processing, 1997. **Proceedings…** Berlin. Spring-Verlag, 1997. p.1-34. (Lecture Notes in Computer Science, v.1291).

37. FEITELSON, Dror; JETTE, M. A. Improved Utilization an Responsiveness with *Gang Scheduling*. In: Job Scheduling Strategies for Parallel Processing, 1997. **Proceedings…** Berlin. Spring-Verlag, 1997. p.238-261. (Lecture Notes in Computer Science, v.1291).
38. FLYNN, M. J. Very High-Speeed Computing Systems. **Proceedings of IEEE**, New York, v.54, n.12, p1901-1909. Dec 1966.
39. GALILÉE, François; CAVALHEIRO, Gerson; ROCH, Jean-Louis; DOREILLE, Mathias. Athapascan-1: on-line building data flow graph in a parallel language. **Proceedings…** PACT 98, Paris, 1998.
40. GEYER, Cláudio F. R. et al. The APPELO Project - Parallel Environment for Logic Programming. In: PROTEM-CC'99 Projects Evaluation Workshop Fase III, 1999. **Proceedings...** Rio de Janeiro, CNPQ, 1999. p.421-454.
41. HILLIS, W.D.; TUCKER, L.W. The CM-5 Connection Machine: A Scalable Supercomputer. **Communications of the ACM**. New York, v.36, n.11, p.31-40. Nov 1993.
42. HOFMANN, M. Dal Cin et al. MEMSY: A Modular Expandable Multiprocessor System. In: Parallel Computer Architectures. 1993. **Proceedings…** Berlin. Spring-Verlag, 1993. p.15-30 (Lecture Notes in Computer Science, v.732).
43. HORI, A. et al. Implementation of *Gang Scheduling* on Workstation Cluster. In: Job Scheduling Strategies for Parallel Processing, 1996. **Proceedings…** Berlin. Spring-Verlag, 1996. p.126-139. (Lecture Notes in Computer Science, v.1162).
44. HUI, Chi-Chung; CHANSON, Samuel. Improved Strategies for Dynamic Load Balancing. **IEEE Concurrency**. New York, v.7, n.3, p.58-67. Jul-Sept 1999.
45. HWANG, K.; XU, Z. **Scalable Parallel Computing: Technology, Architecture**, Programming. McGraw-Hill, New York, 1998.
46. HWANG, Kai. Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill, Inc. 1993. 771p.
47. JENG, M.; SIEGEL, H. A Distributed Management Scheme for Partitionable Parallel Computers. **IEEE Transactions on Parallel and Distributed Systems**. New York, v.1, n.1, p.120-126. Jan 1990.
48. KAFIL, Muhammad; AHMAD, Ishfaq. Optimal Taks Assignment in Heterogeneous Distributed Computing Systems. **IEEE Concurrency**. New York, p43-51, July-September 1998.
49. KRUEGER, P.; SHIVARATRI, N.G. Adaptative Location Policies for Global Scheduling. **IEEE Transactions on Software Enginnering.** New York, v.20, n.6, p.432-444. June 1994.
50. KWO, Yu-Kwong; AHMAD, Ishfaq. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. **Journal of Parallel and Distributed Computing**. New York, v.59, n.3, p.381-422. Nov. 1999.
51. LEISERSON, C. et al. The Network Architecture of the Connection Machine CM-5. **Journal of Parallel and Distributed Computing**. New York, v.33, n.2, p.154-158. Mar 1996.
52. LEUZE, M. R.; DOWDY, L. W.; PARK, K. H. Multiprogramming a Distributed-Memory Multiprocessor. **Concurrency – Prat. & Exp**. New York, v.1, n.1, p.19-33. Sept 1989.
53. LI, K.; CHENG, K. A Two-Dimensional Buddy System for Dynamic Resource Alocation in a Partitionable Mesh Connected System. **Journal of Parallel and Distributed Computing**. New York, v.12, p.79-83. 1991.
54. LIU, T. et al. Non-Contíguos Processor Allocation Algorithms for Mesh-Connected Multicomputers. **IEEE Transactions on Parallel and Distributed Systems**. v.7, n.7, p.712-726. Jul 1997.
55. LOVEMAN, D.B. High Performance Fortran. **IEEE Parallel and Distributed Technology**. New York, v.1, n.1, p.25-42. Feb 1993.
56. MAHESWARAN, Muthucumaru et ali. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. **Journal of Parallel and Distributed Computing.** New York, n.59, p107-131. November 1999.
57. MAUNEY, J.et al. Computational Models and Resource Allocation for Supercomputers. **Proceddings of IEEE**. New York, v.77, n.12, p.1859-1874. Dec 1989.

58. MAY, D; SHEPHERD,R.; KEANE, C. Communicating Process Architecture: Transputers and Occam. In: Future Parallel Computers. 1987. **Proceedings…** Berlin. Spring-Verlag, 1987. p.35-81 (Lecture Notes in Computer Science, v.272).
59. McCANN, C; VASWANI, R; ZAHORJAN, J. A Dynamic Processor Alocation Policy for Multi-Programmed Shared-Memory Multiprocessors. **ACM Transactions on Computing Systems**. v.11, n.2, p.146-178. May 1993.
60. MEHROTRA, P. Data Parallel Programming: The Promises and Limitations of High Performance Fortran. In: Lecture Notes in Computer Science, 1993, **Proceedings...** Berlin: Springer-Verlag, 1993. p.114-114. v.734.
61. MOREIRA, J. E. **On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors**. Department of Electrical and Computer Engineering, Univ. of Illinois at Urbana-Champaign, 1995. PhD. Thesis. Disponível em http://www.csrd.uiuc.edu/reports/1372.ps.gz
62. NASCIMENTO, Saulo; HENRIQUES, Marco. Um Algoritmo Distribuído para Detecção e Localização de Falhas em Redes tipo Hipercubo. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 11., Out. 1999, Natal. **Anais...** Natal: SBC, 1999. 324p. p.133-138.
63. NGUYEN, Thu D.; VASWANI, Raj; ZAHORJAN. Parallel Application Characteristics for Multiprocessor Scheduling Policy Design. In: Job Scheduling Strategies for Parallel Processing, 1996. **Proceedings…** Berlin. Spring-Verlag, 1996. p.175-199. (Lecture Notes in Computer Science, v.1162).
64. NOGUEIRA, Mauro Lúcio B. Robin Hood: **Um Ambiente para a Avaliação de Políticas de Balanceamento de Carga**. Porto Alegre, CPGCC-UFRGS, 1997. 128p. (Dissertação de Mestrado).
65. PADHYE, J; DOWDY, L. Dynamic Versus Adaptive Processor Allocation Policies for Message Passing Parallel Computers: Am Empirical Comparison. In: Job Scheduling Strategies for Parallel Processing, 1996. **Proceedings…** Berlin. Spring-Verlag, 1996. p.224-243. (Lecture Notes in Computer Science, v.1162).
66. PRUYNE, J.; LIVNY, M. Managing Checkpoints for Parallel Programs. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.259-278 (Lecture Notes in Computer Science, v.949).
67. RAI, S. et al. Processor Allocation in Hipercube Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems**. v.6, n.6, p.606-616. Jun 1995.
68. RAMME, F.; RÖMKE, T.; KREMER, K. A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems. In: High-Performance Computing and Networking, 1994. **Proceedings...** Berlin. Spring-Verlag, 1994. p.129-136 (Lecture Notes in Computer Science, v.797).
69. RINARD, Martin; LAM, Monica. The Design, Implementation, and Evaluation of Jade. ACM Transactions on Programming Languages and Systems. v.20, n.3, p483-545. May 1998.
70. ROSTI, E. et al. Analysis of Non-Work-Conserving Processor Partitioning Policies. In: Joint international conference on Measurement and modeling of computer systems. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.165-178 (Lecture Notes in Computer Science, v.949).
71. ROSTI, Emilia et al. Robust Partitioning Schemes of Multiprocessor Systems. **Performance Evaluation**. New York. v. 19, n.2-3, p.141-165. Mar 1994. Disponível em: http://www.cs.wm.edu/~esmirni/ docs/perfeval.ps.gz. Acesso em 16 abril.2000.
72. RUDOLPH, L. et al. Runtime support for parallel language constructs in a tightly coupled multiprocessor. Institute of Computer Science, The Hebrew University, Jerusalem, 1993. (**Technical Report**) Disponível em http://www.cs.huji.ac.il/~moshe/papers/maxi.ps.gz
73. SAPHIR, W; TANNER, L.; TRAVERSAT, B. Job Management Requirements for NAS Parallel Systems and Clusters. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.319-336 (Lecture Notes in Computer Science, v.949).
74. SCHRÖDER, W. PEACE – A Software Backplane for Parallel Computing. **Parallel Computing**. New York, v.20, n10-11, p.1471-1485. Nov 1994.
75. SEITZ, C. L. The Cosmic Cube. **Communications of ACM**. New York, v.28, n.1, p.22-33. Jan 1985.

76. SETIA, Sanjeev. Trace-driven Analysis of Migration-based *Gang Scheduling* Policies for Parallel Computers. **In International Conference on Parallel Processing**, August 1997. Disponível em: http://www.cs.gmu.edu/~setia/papers/gang.ps.gz. Acesso em 07 junho.2000.
77. SGALL, J. On-line Scheduling - a Survey. In A. Fiat and G. Woeginger, editors, On-Line Algorithms, **Lecture Notes in Computer Science. Springer-Verlag**, Berlin, 1997. Disponível em: http://www.math.cas.cz/~sgall/ps /schsurv.ps.gz. Acesso em 13 fev.2000.
78. SHCWIEGELSHOHN, Uwe; RAMIM, Yahyapour. Improving First-Come-First-Serve Job Scheduling by *Gang Scheduling*. In: Job Scheduling Strategies for Parallel Processing, 1998. **Proceedings…** Berlin. Spring-Verlag, 1998. p.180-198. (Lecture Notes in Computer Science, v.1459).
79. SILVA, Fabrício; SCHERSON, Isaac. Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, 11., Out. 1999, Natal. **Anais...** Natal: SBC, 1999. 324p. p.243-247.
80. SILVA, Fabrício; SCHERSON, Isaac. Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang. **Proceedings…** of the 14th International Parallel & Distributed Processing Symposium – IPDPS 2000, Cancun, Mexico, IEEE, 2000. 842p. p.121-126.
81. SMIRNI E., ROSTI E., DOWDY L.W., SERAZZI G. A Methodology for the Evaluation of Multiprocessor Non-Preemptive Allocation Policies. **Journal of Systems Architecture**. v.44, n.9-10, p.703-721. 1998. Disponível em: http://www.cs.wm.edu/~esmirni /docs/jsa.ps.gz. Acesso em 16 abril.2000.
82. SMITH, Warren; FOSTER, Ian; TAYLOR, Valerie. Predicting Application Run Times Using Historical Information. In: Job Scheduling Strategies for Parallel Processing, 1998. **Proceedings…** Berlin. Spring-Verlag, 1998. p.122-142. (Lecture Notes in Computer Science, v.1459).
83. SMITH, Warren; FOSTER, Ian; TAYLOR, Valerie. Scheduling with Advanced Reservations. **Proceedings…** of the 14th International Parallel & Distributed Processing Symposium – IPDPS 2000, Cancun, Mexico, IEEE, 2000. 842p. p.127-132.
84. SOBALVARRO, P. G.; WEIHL, E. Demand-Based CoScheduling of Parallel Jobs on Multiprogrammed Multiprocessors. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.106-126 (Lecture Notes in Computer Science, v.949).
85. SQUILLANTE, M. S. On the Benefits and Limitations of Dynamic Partitioning in Parallel Computer Systems. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.219-238 (Lecture Notes in Computer Science, v.949).
86. THAKUR, Rajeev; CHOUDHARY, Alok; RAMANUJAM, J. Efficient Algorithms for Array Redistribution. **IEEE Transactionson Parallel and Distributed Systems**. New York, v.6, n.7, p.587-594, june 1996.
87. TORELLAS, J. et al. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. **Journal of Parallel and Distributed Computing**. New York, v.24, n.2, p.139-151. Feb 1995.
88. TRYSTRAM, Denis. Scheduling for Parallel and Distributed Systems. In: International School on Advanced Algorithmic Techniques for Parallel Computation with Applications. Natal, 1999. 17p.
89. VAN DER PAS, R. J.; van KATS, J. M. Parallelism in a Multi-user environment. **Parallel Computing**. New York, v.17, n.2, p.185-196. Jun 1991.
90. WALDSPURGER, C. A. et al. Spaw: A Distributed Computational Economy. **IEEE Transactions on Software Enginnering**. New York, v.18, n.2, p.103-177. Feb 1992.
91. WANG, F.; PAPAEFTHYMIOU, M.; SQUILLANTE, M. S. Performance evaluation of *Gang Scheduling* for parallel and distributed multiprogramming. In: Job Scheduling Strategies for Parallel Processing, 1997. **Proceedings…** Berlin. Spring-Verlag, 1997. p.277-298. (Lecture Notes in Computer Science, v.1291).
92. WANG, W. T.; MORRIS, R. J. Load Sharing in Distributed Systems. **IEEE Transaction on Computers**. New York, v.c-34, n.3, p.204-217. Mar 1985.

93. WATTS, Jerrel; TAYLOR, Stephen. A Pratical Approach to Dynamic Load Balancing. **IEEE Transactions on Parallel and Distributed Systems**. New York, v.9, n.3, p.235-248. March 1998.
94. YAMIN, Adenauer C.; **Um Ambiente Para Exploração de Paralelismo na Programação em Lógica.** Porto Alegre: CPGCC da UFRGS, 1994. 204p. Dissertação de Mestrado.
95. YAMIN, Adenauer Corrêa. **Um Estudo das Potencialidades e Limites na Exploração do Paralelismo**. Porto Alegre: PPGC-UFRGS, 1999. 80 p. Trabalho Individual.
96. YANG, Q.; WANG, H. A New Graph Approach to Minimizing Processor Fragmentation in Hypercube Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems**. New York, v.4, n.10, p.1165-1171. Oct 1993.
97. YUE, K. K.; LILJA, D. J. Loop-Level Process Control: an Effective Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. 1995. **Proceedings...** Berlin. Spring-Verlag, 1995. p.182-199 (Lecture Notes in Computer Science, v.949).
98. ZELLNER, Markus. Porting Programs from the iPSC/860 to the AP1000. Australian National University: Computer Science Departament, May 1992 (**Technical Report**). Disponível em: http://cs.anu.edu.au/techreports/1992/TR-CS-92-04.ps.gz. Acesso em 16 abril.2000.
99. ZHANG, Y et al. Improving Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. **Proceedings…** of the 14th International Parallel & Distributed Processing Symposium – IPDPS 2000, Cancun, Mexico, IEEE, 2000. 842p. p.133-142.
100. ZHOU, S. et al. Utopia: A Load-Sharing Facility for Large, Heterogeneous Distributed Computer Systems. **Software: Pratice an Experience**. New York, v.23, n.12, p.1305-1336. Dec 1993.
101. ZHU, Y. H. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. **Journal of Parallel and Distributed Computing**. New York, v.16, p.328-337. 1992.
102. ZOMAYA, Albert. **Parallel & Distributed Computing Handbook**. New York: McGraw Hill, 1996. 1232p.
103. ZOMAYA, Albert; WARD, Chris; MACEY, Ben. Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues. **IEEE Transactions on Parallel and Distributed Systems.** New York, v.10, n.8, p.795-812. August 1999.