Uma Heurística de Agrupamento de Caminhos
para Escalonamento de Tarefas
em Grades Computacionais

Luiz Fernando Bittencourt

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Heurística de Agrupamento de Caminhos para Escalonamento de Tarefas em Grades Computacionais

Luiz Fernando Bittencourt[1]

Março de 2006

Banca Examinadora:

- Edmundo Roberto Mauro Madeira (Orientador)
- Alfredo Goldman vel Lejbman
  DCC-IME-USP
- Luiz Eduardo Buzato
  IC-UNICAMP
- Ricardo de Oliveira Anido (Suplente)
  IC-UNICAMP

[1]**Supported by CNPq, under grant 131586/2004-1.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Bittencourt, Luiz Fernando

B548h Uma heurística de agrupamento de caminhos para escalonamento de tarefas em grades computacionais / Luiz Fernando Bittencourt -- Campinas, [S.P. :s.n.], 2006.

Orientador : Edmundo Roberto Mauro Madeira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas distribuídos. 2. Grades computacionais. 3. Fluxo de trabalho. 4. Processamento distribuído. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: A path clustering heuristic for scheduling task graphs onto a grid

Palavras-chave em inglês (Keywords): 1. Distributed systems. 2. Computational grids (Computer systems). 3. Workflow. 4. Distributed processing.

Área de concentração: Sistemas de computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)
Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)
Prof. Dr. Alfredo Goldman vel Lejbman (IME-USP)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)

Data da defesa: 15/03/06

Uma Heurística de Agrupamento de Caminhos para Escalonamento de Tarefas em Grades Computacionais

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Luiz Fernando Bittencourt e aprovada pela Banca Examinadora.

Campinas, 15 de março de 2006.

Edmundo Roberto Mauro Madeira
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Substitua pela folha com a assinatura da banca

Resumo

Uma grade computacional é um sistema heterogêneo colaborativo, geograficamente distribuído, multi-institucional e dinâmico, onde qualquer recurso computacional ligado a uma rede, local ou não, é um potencial colaborador. Grades computacionais são atualmente um grande foco de estudos relacionados à execução de aplicações paralelas, tanto aquelas que demandam grande poder computacional quanto aquelas que se adaptam bem a ambientes distribuídos. Como os recursos de uma grade pertencem a vários domínios administrativos diferentes com políticas diferentes, cada recurso tem autonomia para participar ou deixar de participar da grade em qualquer momento. Essa característica dinâmica e a heterogeneidade tornam o escalonamento de aplicações, a gerência de recursos e a tolerância a falhas grandes desafios nesses sistemas. Particularmente, o escalonamento desempenha um papel de suma importância, pois é determinante no tempo de execução das aplicações. O escalonamento de tarefas é um problema NP-Completo [6], o que levou ao desenvolvimento de uma heurística para o problema de otimização associado. Neste trabalho apresentamos um escalonador de tarefas em grades computacionais baseado no Xavantes [3], um middleware que oferece suporte a execução de tarefas dependentes através de estruturas de controle hierárquicas chamadas *controladores*. O algoritmo desenvolvido, chamado de *Path Clustering Heuristic* (PCH), agrupa as tarefas com o objetivo de minimizar a comunicação entre os controladores e as tarefas, diminuindo o tempo de execução total do processo.

Abstract

A computational grid is a collaborative heterogeneous, geographically distributed, multi-institutional and dynamic system, where any computational resource with a network connection, local or remote, is a potential collaborator. In computational grids, problems related to the execution of parallel applications, those which need a lot of computational power, as well as those which fit well in distributed environments, are wide studied nowadays. As the grid resources belong to various different administrative domains with different policies, each resource has the autonomy to participate or leave the grid at any time. These dynamic and heterogeneous characteristics make the application scheduling, the resource management and the fault tolerance relevant issues on these systems. Particularly, the scheduler plays an important role, since it is determinative in the execution time of an application. The task scheduling problem is NP-Complete [6], what led to the development of a heuristic for the associated optimization problem. In this work we present a task scheduler for a computational grid based on Xavantes [3], a middleware that supports dependent task execution through control structures called *controllers*. The developed algorithm, called *Path Clustering Heuristic* (PCH), clusterizes tasks aiming to minimize the communication between controllers and tasks, reducing the process execution time.

Agradecimentos

Agradeço primeiramente à minha família, principalmente aos meus pais, Luiz Carlos e Maria do Rocio, pelo apoio incondicional às decisões que tomei até aqui, além do carinho e dedicação comigo e com meus irmãos, Cíntia e Emerson.

Ao meu orientador, Professor Edmundo Madeira, pelas dicas preciosas e pela orientação sempre paciente, atenciosa e cuidadosa, ao Fábio Cicerre e ao Professor Luiz Buzato, pelo trabalho em conjunto. Agradeço também aos professores Alfredo Goldman vel Lejbman (IME-USP) e Ricardo de Oliveira Anido, pelas sugestões em relação à esta dissertação e trabalhos futuros.

À minha namorada, Nayrane Caldeira, pelo companheirismo e pela compreensão nos últimos dois anos. Às suas irmãs Keyse e Shayenne, e à sua mãe Suely, pela amizade e pelos momentos agradáveis.

Aos meus amigos aqui em Campinas: do Instituto de Computação, do Laboratório de Sistemas Distribuídos (LSD), do Laboratório de Alto Desempenho (LAD, *aka* Laboratório de Otimização Combinatória (LOCO) ou Laboratório de Uso Geral (LUG)) e das salas espalhadas pelos prédios do instituto. Obrigado pelos momentos de descontração em repúblicas e bares, pelos bate-papos no café e nos corredores, pelos momentos desportivos, pelas dicas de corredor, pelas dicas de não-corredor e derivados. Allan, André Vignatti (meu companheiro de graduação, e hoje de pós-graduação e de quarto), André Atanásio, Augusto, Borin, Bruno, Carlos Eduardo, Carlos Senna, Cláudio, Daniel Batista, Daniel Manzato, Diogo, Evandro, Jurandy, Leonardo Rangel, Leonardo Tizzei, Leonel, Leyza, Luís Meira, Mamão, Neumar, Pará, Patrick, Sheila, Tiago Coelho, Tiago Moronte. Aos vizinhos Cristiano, Larissa, Natália e Rafael Gazela. À Maíra, por indução em André Vignatti, seu namorado. Devo incluir aqui também agradecimentos ao DCE, ao Instituto de Artes e ao Instituto de Filosofia e Ciências Humanas pelas festas no campus.

Aos meus amigos de Curitiba e da UFPR, que tornam as visitas à minha cidade natal sempre tão prazerosas. Por ter menos contato agora que antes, não vou citar nomes para não deixar nenhum de fora. Todos estão implicitamente incluídos aqui.

Aos professores e funcionários do Instituto de Computação com os quais tive contato, pelos ensinamentos e pela atenção. Aos professores do Departamento de Informática da

Universidade Federal do Paraná, pela qualidade das aulas do curso de bacharelado em ciência da computação. Ao professor Jair Donadelli Jr., meu orientador no trabalho de conclusão de curso na UFPR. Aos professores Hélio Pedrini e Elias Procópio, também da UFPR, pelas dicas e pelo incentivo em relação à mudança para outra cidade. Aos professores Fábio de Oliveira Pedrosa e Leonardo Magalhães Cruz e ao CNPq, pela oportunidade de realizar uma iniciação científica.

Ao CNPq, pelo apoio financeiro, e à parcela honesta dos contribuintes brasileiros, que permite o financiamento científico neste país.

Se nestes agradecimentos não existe nenhuma referência à você que os está lendo, existem quatro explicações para esse fenômeno. A primeira é o fato de sua contribuição ter sido sutil e eu não a percebi. A segunda, e mais provável, é eu ter esquecido. A terceira é porque eu não te conheço. A quarta... ah, deixa pra lá! Assim, caso você tenha motivos para acreditar que contribuiu de alguma forma com este trabalho, permito-lhe escrever de próprio punho seu nome nas linhas em branco seguintes.

Quidquid latine dictum sit, altum sonatur.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

ix

1 Introdução **1**

2 Conceitos Básicos **5**
2.1 Computação em Grade 5
2.2 Escalonamento 6
2.3 Escalonamento de Tarefas 7

3 Trabalhos Relacionados **11**

4 O Middleware Xavantes **15**
4.1 Modelo de Programação 15
4.2 Infra-estrutura 18
4.3 Exemplo de Processo 21

5 Escalonamento no Middleware Xavantes **25**

6 Algoritmo Proposto **33**
6.1 Definições 34
6.2 Seleção de Tarefas e Agrupamento 35
6.2.1 Algoritmo de Seleção de Tarefas e Agrupamento 36
6.2.2 Exemplo de Execução 37
6.3 Seleção de Recursos 37
6.3.1 Algoritmo de Seleção de Recursos 38

6.3.2 Exemplo de Execução 39
6.4 Escalonamento de Controladores 40
6.4.1 Algoritmo de Escalonamento de Controladores 40
6.4.2 Exemplo de Execução 41
6.5 O Algoritmo *Path Clustering Heuristic* 42
6.6 Busca de Recursos em Grupos Adjacentes 43
6.7 Análise de Complexidade 45

7 Resultados Experimentais **55**
7.1 Métricas de Desempenho 55
7.2 O Algoritmo HEFT 56
7.3 O Algoritmo CPOP 57
7.4 Resultados 57
7.4.1 Sistema Heterogêneo 58
7.4.2 Topologia de Grupos 60
7.4.3 Busca de Recursos em Grupos Adjacentes 86
7.4.4 Recursos de Grupos Adjacentes no Repositório Local 90

8 Conclusão **93**

Bibliografia **95**

Lista de Tabelas

6.1 Recursos usados no exemplo da Figura 5.4. A largura de banda entre todos os recursos é fornecida. 37
6.2 Atributos dos nós após executar o algoritmo para o grafo da Figura 5.4. . . 38
6.3 Resultados da seleção de recursos para o grafo da Figura 5.4. 40

7.1 *Makespans* e desvios padrões médios para execuções com comunicação baixa. 81
7.2 *Makespans* e desvios padrões médios para execuções com comunicação média. 83
7.3 *Makespans* e desvios padrões médios para execuções com comunicação alta. 85

Lista de Figuras

2.1 Exemplo de DAG representando um processo. 9

4.1 Controlador seqüencial. 16
4.2 Controladores *par* e *parswitch*. 16
4.3 Controladores *parfor* e *parwhile*. 16
4.4 Exemplo de DAG representando um processo com controladores. 17
4.5 Organização dos recursos no middleware Xavantes. 19
4.6 Interações entre AM, GM e PM para execução de uma tarefa. 20
4.7 Interações entre AM, GMs e PM para execução de uma tarefa em grupo adjacente. 21

5.1 Escalonador na arquitetura. 26
5.2 Grafo aceito pelo modelo de programação. 28
5.3 Grafo não aceito pelo modelo de programação. 30
5.4 Representação completa de um processo. 32

6.1 Escalonamento de tarefas e controladores para o grafo da Figura 5.4 43
6.2 Grafo G': G após adição de um controlador seqüencial com k tarefas. . . . 46
6.3 Grafo G': G após adição de um controlador paralelo com k tarefas. 46
6.4 Grafo G'': G' após adição de um controlador seqüencial com m tarefas. . . 48
6.5 Grafo G'': G' após adição de um controlador seqüencial com m tarefas. . . 48
6.6 Grafo G'': G' após adição de um controlador seqüencial com m tarefas. . . 48
6.7 Grafo G'': G' após adição de um controlador paralelo com m tarefas. . . . 49
6.8 Grafo G'': G' após adição de um controlador paralelo com m tarefas. . . . 50
6.9 Grafo G'': G' após adição de um controlador paralelo com m tarefas. . . . 50

7.1 Número de escalonamentos com menor *makespan*. 59
7.2 *Overhead* de comunicação dos controladores. 60
7.3 SLR médio. 61
7.4 *Speedup* médio. 61

7.5 Número de escalonamentos com menor *makespan* em no máximo 2 grupos com no máximo 3 recursos. . . . 62
7.6 *Overhead* de comunicação dos controladores em no máximo 2 grupos com no máximo 3 recursos. . . . 63
7.7 SLR médio em no máximo 2 grupos com no máximo 3 recursos. . . . 64
7.8 *Speedup* médio em no máximo 2 grupos com no máximo 3 recursos. . . . 64
7.9 Número de escalonamentos com menor *makespan* em no máximo 2 grupos com no máximo 10 recursos. . . . 65
7.10 *Overhead* de comunicação dos controladores em no máximo 2 grupos com no máximo 10 recursos. . . . 66
7.11 SLR médio em no máximo 2 grupos com no máximo 10 recursos. . . . 66
7.12 *Speedup* médio em no máximo 2 grupos com no máximo 10 recursos. . . . 67
7.13 Número de escalonamentos com menor *makespan* em no máximo 3 grupos com no máximo 3 recursos. . . . 68
7.14 *Overhead* de comunicação dos controladores em no máximo 3 grupos com no máximo 3 recursos. . . . 68
7.15 SLR médio em no máximo 3 grupos com no máximo 3 recursos. . . . 69
7.16 *Speedup* médio em no máximo 3 grupos com no máximo 3 recursos. . . . 70
7.17 Número de escalonamentos com menor *makespan* em no máximo 4 grupos com no máximo 5 recursos. . . . 70
7.18 *Overhead* de comunicação dos controladores em no máximo 4 grupos com no máximo 5 recursos. . . . 71
7.19 SLR médio em no máximo 4 grupos com no máximo 5 recursos. . . . 71
7.20 *Speedup* médio em no máximo 4 grupos com no máximo 5 recursos. . . . 72
7.21 Número de escalonamentos com menor *makespan* em no máximo 5 grupos com no máximo 3 recursos. . . . 73
7.22 *Overhead* de comunicação dos controladores em no máximo 5 grupos com no máximo 3 recursos. . . . 73
7.23 SLR médio em no máximo 5 grupos com no máximo 3 recursos. . . . 74
7.24 *Speedup* médio em no máximo 5 grupos com no máximo 3 recursos. . . . 75
7.25 Número de escalonamentos com menor *makespan* em no máximo 8 grupos com no máximo 3 recursos. . . . 75
7.26 *Overhead* de comunicação dos controladores em no máximo 8 grupos com no máximo 3 recursos. . . . 76
7.27 SLR médio em no máximo 8 grupos com no máximo 3 recursos. . . . 76
7.28 *Speedup* médio em no máximo 8 grupos com no máximo 3 recursos. . . . 77
7.29 Número de escalonamentos com menor *makespan* em no máximo 8 grupos com no máximo 10 recursos. . . . 78

7.30 *Overhead* de comunicação dos controladores em no máximo 8 grupos com no máximo 10 recursos. 78
7.31 SLR médio em no máximo 8 grupos com no máximo 10 recursos. 79
7.32 *Speedup* médio em no máximo 8 grupos com no máximo 10 recursos. 80
7.33 *Makespan* médio sobre todas as execuções com comunicação baixa. 80
7.34 Desvio padrão médio para execuções com comunicação baixa. 81
7.35 *Makespan* médio sobre todas as execuções com comunicação média. 82
7.36 Desvio padrão médio para execuções com comunicação média. 82
7.37 *Makespan* médio sobre todas as execuções com comunicação alta. 84
7.38 Desvio padrão médio para execuções com comunicação alta. 84
7.39 Número de escalonamentos com menor *makespan* em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos. 87
7.40 *Overhead* de comunicação dos controladores em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos. 88
7.41 SLR médio em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos. 88
7.42 *Speedup* médio em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos. 89
7.43 Número de melhores escalonamentos de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo. 90
7.44 SLR médio de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo. 91
7.45 *Speedup* médio de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo. 92

Lista de Algoritmos

1 Processo Xavantes 22
2 Atividade Xavantes 23
3 Código para Figura 5.2 29
4 Código para Figura 5.3 31
5 PCH - visão geral 33
6 gera_próximo_agrupamento 36
7 seleciona_melhor_recurso 39
8 escalona_controladores 42
9 PCH 43
10 PCH com busca de recursos 45
11 HEFT 57
12 CPOP 58

Capítulo 1

Introdução

A computação é uma grande aliada na resolução de problemas que necessitam grande quantidade de processamento. À medida que a tecnologia avança, temos computadores com maior poder de processamento, maior quantidade de memória e maior capacidade de armazenamento. Contudo, esse avanço não é suficiente para suprir as necessidades de todas as aplicações, as quais também têm uma demanda crescente por computação. Sistemas distribuídos resultam da integração de recursos computacionais de várias fontes para aumentar a capacidade de processamento, a quantidade de memória e a capacidade de armazenamento, contribuindo para a diminuição do tempo de execução de aplicações. A execução de uma tarefa paralelamente em vários computadores é conhecida como *computação distribuída*.

Sistemas distribuídos vêm substituindo os supercomputadores na execução de aplicações que demandam alto poder de processamento, grande quantidade de memória ou alta capacidade de armazenamento. O desenvolvimento da tecnologia de redes locais permitiu a criação de *clusters* de computadores homogêneos, que se tornaram uma alternativa de baixo custo e alto desempenho. As constantes evoluções da tecnologia de redes, dos sistemas operacionais e das linguagens de programação permitiram a criação de sistemas heterogêneos, transformando qualquer recurso computacional em um potencial colaborador em sistemas distribuídos para execução de aplicações paralelas.

Com a possibilidade de comunicação entre recursos computacionais heterogêneos, a partir da premissa de que qualquer recurso pode ser agregado a um sistema, em meados dos anos 90 foi proposto o conceito de *grades computacionais* [9]. Esse conceito apresenta como objetivo o compartilhamento coordenado de recursos heterogêneos, geograficamente distantes, em larga escala. Esses recursos podem ser variados, tais como processadores, armazenamento em disco, hardware especializado e softwares.

Uma grade tem potencialmente um grande poder de processamento, pois pode ser composta por todos os recursos computacionais interligados de qualquer forma por alguma

rede. Um middleware para grades computacionais deve fornecer suporte para a execução de aplicações de tipos e tamanhos variados em um ambiente heterogêneo e dinâmico. Oferecer bom desempenho e confiabilidade nesse ambiente não é de forma alguma trivial, já que o desempenho e a disponibilidade de cada recurso pode variar imprevisivelmente. Alguns serviços imprescindíveis para o funcionamento de uma grade computacional são os serviços de escalonamento, monitoramento de recursos, gerência de recursos, controle de acesso e gerenciamento de processos. O tamanho de uma grade pode variar muito em pouco tempo, então esses serviços e o middleware como um todo devem ser escaláveis, além de independentes de plataforma.

Escalonamento é fundamental para atingir bom desempenho na execução de processos em sistemas distribuídos. Em uma grade computacional podem ser executadas tarefas acopladas, as quais necessitam de comunicação entre si. Devido a esse acoplamento, tarefas podem ser estruturadas como workflows com as informações necessárias para que o escalonador distribua as tarefas eficientemente. O problema de escalonamento de tarefas é, em geral, NP-Completo [6], levando ao desenvolvimento de algoritmos de escalonamento baseados em heurísticas. Em um sistema heterogêneo e dinâmico como uma grade computacional, esse problema adquire novas variáveis e torna-se ainda mais complexo.

O escalonador aqui apresentado foi desenvolvido para trabalhar com o Xavantes [3], um middleware que oferece suporte à execução de tarefas dependentes através de estruturas de controle hierárquicas chamadas *controladores*. No Xavantes os recursos da grade são estruturados em grupos, com o objetivo de executar atividades dependentes em recursos próximos, de forma a diminuir a comunicação, aumentar o desempenho geral na execução dos processos e melhorar a confiabilidade. Além disso, como os recursos são distribuídos em grupos, podemos ter escalonadores independentes para cada grupo, evitando os problemas dos escalonadores centralizados ou totalmente distribuídos. O modelo de programação do Xavantes [3] estrutura a execução dos processos de forma hierárquica, promovendo controle de fluxo estruturado, e facilitando a especificação, monitoramento e coordenação da execução das atividades componentes de um processo.

A heurística desenvolvida para promover o escalonamento leva em consideração os controladores, pois a comunicação entre tarefas deve ser feita através deles. Dessa forma, a seleção de tarefas durante a composição de um agrupamento tenta minimizar a comunicação entre tarefas e controladores, para que a execução do processo não seja prejudicada por esse tipo de comunicação. Cada agrupamento é executado em um recurso, selecionado de acordo com a política de seleção de recursos desenvolvida. Escalonadas todas as tarefas, os controladores devem ser também escalonados. Os controladores são escalonados em recursos que minimizam a comunicação com as tarefas, para não tornar o tempo de comunicação um fator impeditivo na execução do processo. As simulações conduzidas mostram que o algoritmo fornece bom desempenho e, em conjunto com o mid-

dleware, provê confiabilidade, alta disponibilidade, escalabilidade e recuperação de falhas para execução de processos em um ambiente de grade computacional.

No Capítulo 2 apresentamos os conceitos básicos de computação em grade e dos problemas de escalonamento, necessários ao entendimento do algoritmo proposto e seu objetivo. O Capítulo 3 apresenta trabalhos relacionados à área de escalonamento de tarefas em sistemas homogêneos e heterogêneos. O middleware Xavantes, para o qual o algoritmo proposto foi desenvolvido, é detalhado no Capítulo 4, mostrando o modelo de programação e a infra-estrutura. O escalonamento dentro do Xavantes é discutido em detalhes no Capítulo 5. No Capítulo 6, o algoritmo proposto é apresentado e são comentados os passos de seleção e agrupamento de tarefas, seleção de recursos, escalonamento de controladores e a análise de complexidade do algoritmo. Resultados experimentais são mostrados no Capítulo 7, comparando o algoritmo proposto com outros dois algoritmos existentes. Finalmente, o Capítulo 8 apresenta as considerações finais e uma visão geral de trabalhos futuros.

Capítulo 2

Conceitos Básicos

Conceitos básicos de computação em grade e escalonamento são apresentados neste capítulo, iniciando com a definição de computação em grade e suas características principais. A seguir são introduzidos os conceitos de escalonamento e escalonamento de tarefas, temas intrínsecos a este trabalho.

2.1 Computação em Grade

Computação em grade consiste no compartilhamento coordenado de recursos por *organizações virtuais* multi-institucionais e dinâmicas, onde uma organização virtual é um conjunto de indivíduos ou instituições que fazem o compartilhamento de recursos coordenado e controlado, com fornecedores e consumidores de recursos que definem claramente o que pode ser compartilhado e sob que condições esse compartilhamento ocorre [10]. Grades computacionais são atualmente um grande foco de estudos relacionados à execução de aplicações paralelas, tanto aquelas que demandam alto poder de processamento quanto aquelas que se adaptam bem a ambientes distribuídos.

Uma grade computacional é um sistema distribuído heterogêneo que possui uma estrutura de gerenciamento e coordenação que controla a execução de tarefas, a disponibilidade dos recursos e os serviços que fazem parte da infra-estrutura. Por se tratar de um sistema heterogêneo, a estrutura de gerenciamento deve manter informações sobre a capacidade dos recursos disponíveis, além de informações sobre a utilização desses recursos e suas políticas de acesso. A execução de tarefas deve ser coordenada de forma a otimizar a utilização dos recursos e minimizar o tempo de execução das aplicações, obedecendo critérios necessários à execução de cada tarefa e as dependências de dados existentes.

Podemos citar como características de uma grade computacional:

- São grandes em termos de recursos potencialmente disponíveis;

- São distribuídas: latências substanciais em movimentação de dados podem ser significativas em relação ao tempo de execução de uma aplicação. Esse fator deve ser considerado pelo escalonador para garantir que o tempo de comunicação não seja proibitivo;
- São heterogêneas: capacidades e propriedades dos nós podem ser significativamente diferentes;
- Estão além dos limites das organizações: nós diferentes podem ter diferentes políticas de acesso.

Organizações virtuais podem variar em tamanho, propósito, estrutura e escopo, pois são baseadas em um compartilhamento *dinâmico* de recursos. Isso significa que a qualquer momento um recurso que faz parte da organização virtual pode se retirar dela, seja por um problema técnico ou pela necessidade de que o recurso seja utilizado para outro propósito. Além disso, o compartilhamento de recursos é condicional, isto é, o administrador do recurso o disponibiliza sujeito a restrições de quem pode utilizá-lo, quando pode ser utilizado e o que pode ser utilizado.

Um conjunto de organizações virtuais forma uma grade computacional. Devem haver protocolos para comunicação entre os coordenadores das organizações virtuais ou um coordenador central, que faz a troca de mensagens entre organizações virtuais. Podemos ter um gerenciador de recursos centralizado, que mantém informações sobre a disponibilidade e capacidade dos recursos da grade, e/ou gerenciadores de grupos de recursos, que trocam informações entre si para se manterem atualizados quanto à quantidade e qualidade dos recursos. Com coordenadores de processo ocorre o mesmo, com a grade contendo um coordenador centralizado e/ou vários coordenadores distribuídos. Cada abordagem tem suas vantagens e desvantagens, podendo cada uma ser utilizada para propósitos diferentes em grades com objetivos distintos.

2.2 Escalonamento

O problema de escalonamento envolve uma aplicação e um ambiente alvo, onde a aplicação será executada. O escalonador é a entidade que manipula a aplicação e atribui cada um de seus componentes a um recurso. Portanto, o tempo de execução de uma aplicação em um sistema distribuído é intrínseco ao escalonamento que lhe for atribuído. O principal objetivo de um escalonador é minimizar o tempo de execução das aplicações (*makespan*), escalonando seus componentes de forma a maximizar o paralelismo na execução das aplicações e minimizar a comunicação, conseqüentemente otimizando a utilização dos recursos.

Uma definição simples para o problema de escalonamento pode ser formulada admitindo-se dois conjuntos. O primeiro conjunto $A = \{a_1, a_2, ..., a_n\}$, composto por aplicações que devem ser executadas em algum recurso, e o segundo conjunto $R = \{r_1, r_2, ..., r_m\}$, composto pelos recursos disponíveis. O objetivo do escalonador é construir um mapeamento de elementos de A em elementos de R, buscando minimizar o tempo de execução dos elementos de A.

Em uma grade computacional um recurso pode ficar indisponível a qualquer momento, o que traz mais problemas quando tratamos de escalonamento nesse tipo de sistema distribuído. Além disso, o poder de processamento e a largura de banda disponíveis em cada recurso variam independentemente dos processos em execução na grade, pois tais recursos estão em domínios administrativos diferentes e podem ser utilizados para a execução de tarefas alheias às aplicações executadas na grade. Essa variação quantitativa e qualitativa de recursos deve ser considerada pelo escalonador, utilizando informações sobre o estado atual do sistema para escolher o recurso em que cada parte de uma aplicação será executada.

Outra característica do escalonador deve ser a escalabilidade. Uma grade computacional pode variar muito de tamanho, sendo pequena em determinados momentos e muito grande em outros. O algoritmo do escalonador deve adaptar-se a essa variação de tamanho para que obtenha bom desempenho em todos os cenários possíveis.

Um escalonador de aplicações em grades computacionais deve considerar a latência na movimentação de dados dentro da grade. A grade pode conter recursos geograficamente muito distantes, o que pode provocar um aumento significativo no tempo demandado para a movimentação tanto dos dados resultantes quanto do código da aplicação. Pode não ser vantajoso executar uma aplicação que seja muito grande ou que tenha resultados grandes em um recurso distante do recurso que submete a aplicação, pois o tempo para movimentar a aplicação e seu resultado pode ser substancialmente superior ao tempo de execução e transmissão de dados em um recurso próximo, mesmo que esse recurso tenha um desempenho computacional inferior.

2.3 Escalonamento de Tarefas

No escalonamento de tarefas uma aplicação (ou processo) é composta de tarefas que têm dependências de dados, onde a execução de cada tarefa deve respeitar as precedências impostas por tais dependências. Então, o escalonador deve tratar das dependências de dados, custos de comunicação entre tarefas e custos de computação das tarefas componentes do processo.

No problema de otimização que tem como objetivo minimizar o *makespan* dos processos compostos por tarefas, o escalonador atribui cada tarefa que faz parte de um processo

a um processador, obedecendo as dependências de dados, de maneira que algumas tarefas possam ser executadas em paralelo. Sendo esse um problema NP-Difícil [6], exceto em alguns casos restritos, existe uma quantidade muito grande de combinações entre tarefas e recursos, tornando inviável a busca exaustiva pela solução ótima para cada processo executado. Um escalonador deve tratar das precedências entre tarefas (dependências de dados), custos de comunicação entre tarefas e custos de execução das tarefas. Em um sistema heterogêneo, cada tarefa pode ter custos de computação e comunicação diferentes em cada recurso, já que cada recurso pode ter capacidade computacional singular.

Um processo composto de tarefas é tipicamente modelado como um grafo acíclico direcionado (directed acyclic graph - DAG). Para cada nó n de um DAG não existe qualquer caminho direcionado que inicia e termina em n. Um processo é representado por um DAG $G = (N, E)$, onde:

- N é o conjunto de $n = |N|$ nós, e $n_i \in N$ representa uma tarefa atômica que deve ser executada em um recurso;
- E é o conjunto de $e = |E|$ arestas direcionadas, onde $e_{i,j} \in E$ representa dependência de dados entre n_i e n_j. Isso implica que n_j não pode iniciar sua execução antes que n_i termine e envie os dados necessários a n_j;
- $suc(n_i)$ é o conjunto de sucessores do nó n_i no grafo acíclico direcionado;
- $pred(n_i)$ é o conjunto de predecessores do nó n_i no grafo acíclico direcionado.

Um nó sem predecessores é chamado de *nó de entrada* e um nó sem sucessores é chamado de *nó de saída*. Esses nós podem ser convenientemente criados com custos de computação e comunicação iguais a zero para que o processo tenha apenas um nó de entrada e um nó de saída. Cada nó (ou tarefa) do grafo é rotulado com o custo de computação e cada aresta é rotulada com o custo de comunicação. Uma tarefa é dita *pronta* se todos os seus predecessores estão escalonados, isto é, n_i é uma tarefa *pronta* se $\{\forall\ n_h \in pred(n_i)\} \Rightarrow \{n_h \text{ está escalonada}\}$.

Essa representação permite a visualização dos componentes que formam a aplicação e como esses componentes interagem entre si. O modelo de programação usado pelo Xavantes permite que essa representação seja utilizada, transformando os processos especificados pelo modelo em DAGs, com estes sendo utilizados como entrada do escalonador. A Figura 2.1 mostra um exemplo de processo representado por um grafo acíclico direcionado. As tarefas são representadas pelos nós e as dependências entre tarefas são representadas pelas arestas. Além da numeração, cada nó possui um rótulo com seu custo computacional. Por exemplo, a tarefa 1 tem custo computacional 3500. Os rótulos das arestas representam o custo de comunicação entre as tarefas. Por exemplo, a comunicação entre os nós 1 e 2

é 200. Como o nó 2 depende dos dados enviados pelo nó 1, o nó 2 só pode iniciar sua execução após o término da execução do nó 1. No caso do nó 11, as dependências de dados podem ser interpretadas de maneiras diferentes. Uma maneira é interpretar a condição para que a tarefa 11 seja executada como um E lógico, isto é, a tarefa 11 só pode iniciar sua execução após o término da execução e recebimento dos dados das tarefas 10, 3 *e* 9. Essa é a forma mais comum de interpretação, e a grande maioria dos algoritmos de escalonamento somente realiza esse tipo de interpretação. Outra maneira é interpretar a condição como um OU lógico, ou seja, a tarefa 11 pode iniciar sua execução assim que uma das três tarefas, 10 *ou* 3 *ou* 9, termine sua execução e envie os dados. Uma terceira maneira é tornar essa condição uma composição de Es e OUs lógicos. Por exemplo, o início da tarefa 11 está sujeito à condição lógica (10 OU 3 E 9). Grafos de processos interpretados dessa maneira são chamados de *grafos condicionais*.

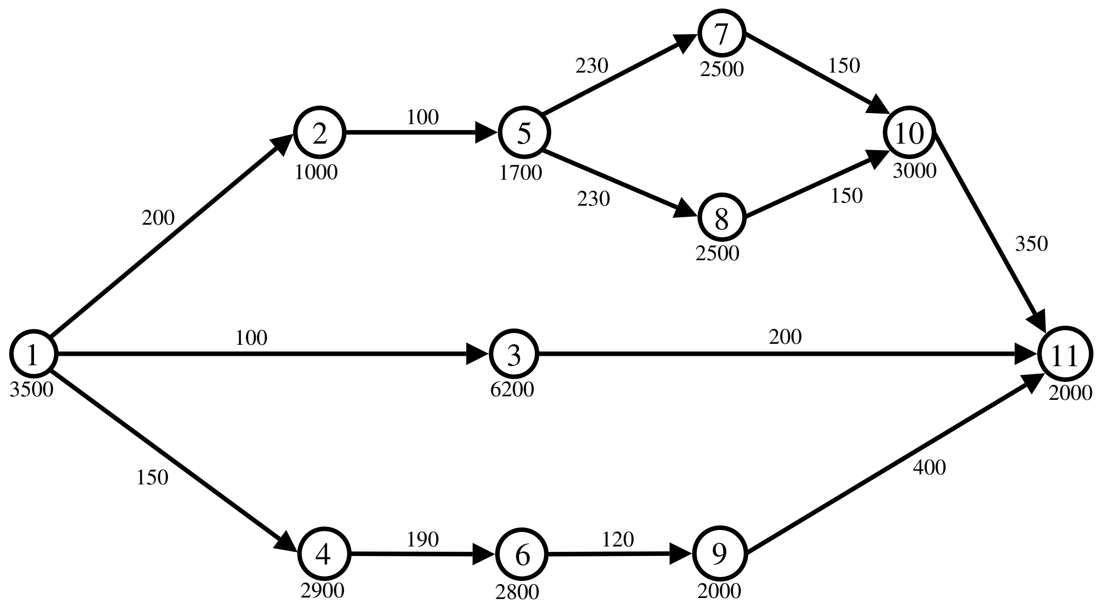

Figura 2.1: Exemplo de DAG representando um processo.

Atrelado ao conceito de grafos condicionais está o conceito de *grafos dinâmicos*. Nestes, ao invés de haver uma condição lógica que determina quando uma tarefa será executada, há uma condição lógica determinando *se* uma tarefa será executada. Nesse tipo de grafo uma tarefa que está inicialmente no DAG pode não ser executada, tornando nulo o tempo de execução da tarefa escalonada.

No escalonamento de tarefas, o ambiente computacional considerado é um conjunto R de $r = |R|$ recursos heterogêneos que podem executar as tarefas representadas pelo DAG. Esses recursos estão interconectados por uma rede com *links* de comunicação heterogêneos e são capazes de transmitir e processar dados simultaneamente, porém só podem executar

uma tarefa de cada vez. O custo de comunicação entre tarefas que têm dependência de dados e que executam no mesmo recurso é igual a zero, pois os dados não precisam percorrer nenhum *link* de comunicação externo ao recurso.

Capítulo 3

Trabalhos Relacionados

O problema de escalonamento de tarefas é NP-Completo. Problemas com essa característica podem ser abordados de diferentes formas, como, por exemplo, utilizando algoritmos de aproximação ou análise combinatória. Entretanto, a abordagem de heurísticas é plena maioria nos esforços relacionados ao desenvolvimento de algoritmos de escalonamento de tarefas. O escalonamento de tarefas é um problema amplamente estudado em sistemas homogêneos [6, 17, 20, 28]. Com o advento dos sistemas heterogêneos, passou a ser necessário considerar a heterogeneidade dos recursos e da largura de banda de comunicação [2, 16, 21, 24, 26].

Uma grade computacional é um sistema heterogêneo, dinâmico e amplamente distribuído. Essas características trazem às grades algumas peculiaridades em relação aos sistemas heterogêneos anteriores, que devem ser tratadas pelo escalonador e/ou pelo middleware. O estudo de questões intrínsecas às grades computacionais tem recebido grande atenção da comunidade científica atualmente [3, 4, 7, 8, 12–14, 29].

O escalonamento de tarefas em grades computacionais é um escalonamento em um sistema heterogêneo. Entretanto, o escalonador e o middleware devem tratar problemas existentes em grades que não eram tratados por escalonadores desenvolvidos para sistemas heterogêneos não dinâmicos e restritamente distribuídos. Nestes, o algoritmo de escalonamento precisa apenas determinar em qual recurso cada tarefa será executada, sem levar em consideração recuperação de falhas e a variação de desempenho dos recursos. *List scheduling* [26], *clustering* [2] e *task duplication* [16] são algumas técnicas utilizadas em algoritmos de escalonamento de tarefas, tanto para sistemas heterogêneos como homogêneos. A técnica de algoritmos genéticos também é utilizada no escalonamento de tarefas em sistemas homogêneos [5, 19] e heterogêneos [15]. Existem também alguns estudos que utilizam técnicas de algoritmos de aproximação [12, 23].

A técnica de ***list scheduling*** consiste em criar uma lista de tarefas a serem escalonadas de acordo com algum critério de prioridade pré-definido. Com essa lista criada, o

algoritmo seleciona uma tarefa da lista, de acordo com uma estratégia de seleção de tarefas, e escalona a tarefa em algum recurso, obedecendo a política de seleção de recursos do algoritmo. O algoritmo *Heterogeneous Earliest Finish Time* (HEFT) [26] utiliza essa técnica, onde o atributo $rank_u$ é calculado percorrendo as arestas do grafo, iniciando no nó de saída e terminando no nó de entrada. O $rank_u$ determina a prioridade de cada tarefa do processo, sendo o critério de seleção de tarefas para escalonamento. Por não se tratar de um algoritmo desenvolvido para grades computacionais, o HEFT não possui nenhuma estratégia para prevenir ou recuperar falhas e, principalmente, não leva em consideração a variação de desempenho dos recursos. Além disso, o escalonamento resultante pode proporcionar uma dispersão de tarefas que não teria desempenho aceitável se considerarmos o uso de controladores.

O algoritmo *Critical Path on a Processor* CPOP [26] também utiliza a técnica de *list scheduling*. O CPOP, além de utilizar o atributo $rank_u$, calcula o atributo $rank_d$ para cada nó, percorrendo o grafo do nó de entrada até o nó de saída. O critério para selecionar qual é a próxima tarefa a ser escalonada pelo CPOP é escolher a tarefa que possui maior P_i, onde $P_i = rank_u + rank_d$. A principal diferença entre CPOP e HEFT é que o primeiro agrupa os nós do caminho crítico do grafo inicial em um mesmo recurso. Como acontece com o HEFT, o CPOP pode causar a dispersão das tarefas do processo, o que não é compatível com a utilização de controladores.

Clustering é uma técnica de duas fases. Na primeira, chamada fase de agrupamento (*clustering*), os nós são agrupados em um sistema homogêneo virtual composto por um número ilimitado de processadores. Tarefas atribuídas ao mesmo *cluster* executarão no mesmo recurso do sistema heterogêneo real. A segunda fase consiste em atribuir cada *cluster* a um processador do sistema heterogêneo real, de acordo com alguma estratégia. O algoritmo *Clustering for Heterogeneous Processors* (CHP) [2] utiliza a técnica de *clustering*. Na fase de agrupamento o CHP cria *clusters* de tarefas assumindo um sistema homogêneo virtual composto por um número ilimitado de processadores com a capacidade de processamento do pior recurso disponível no sistema heterogêneo real. Na fase de mapeamento os *clusters* são atribuídos aos processadores heterogêneos usando uma abordagem baseada na técnica de *list scheduling*. Durante essa fase, o CHP procura pelo recurso que minimiza o tempo estimado de término de todos os *clusters* já mapeados. O CHP também não foi desenvolvido especificamente para grades, o que o leva a ter as mesmas desvantagens apresentadas pelo HEFT.

A técnica de ***task duplication*** consiste em duplicar tarefas, isto é, escaloná-las em mais de um recurso, conseqüentemente executando-as mais de uma vez. O algoritmo *Heterogeneous Critical Nodes with Fast Duplicator* (HCNFD) [16] é baseado nas técnicas de *list scheduling* e *task duplication*. Na fase de criação da lista de tarefas, os nós são inseridos em uma fila usando uma prioridade calculada pelo algoritmo. Na fase de seleção

de recursos, o primeiro nó da fila é selecionado e o algoritmo escolhe o processador que minimiza o tempo estimado de término da tarefa. Após isso, o predecessor dessa tarefa que possuir maior prioridade é duplicado na janela de tempo disponível antes da tarefa, caso essa janela tenha tamanho suficiente para tal e a duplicação reduza o tempo estimado de término calculado anteriormente. Novamente, a falta de consideração em relação aos problemas das grades e aos controladores não torna esse algoritmo aplicável ao middleware Xavantes.

A dinamicidade da grade e sua característica amplamente distribuída requerem que o escalonamento de tarefas tenha uma estratégia que promova também confiabilidade, evitando perda de dados computados e de processamento já realizado, o que não é considerado pelos algoritmos HEFT, CHP e HCNFD. Em [4], são propostas estratégias de escalonamento para o projeto GraDS, incluindo uma abordagem para escalonamento de workflows e re-escalonamento de aplicações. O escalonador de workflows do projeto GraDS utiliza uma estratégia para classificar cada recurso de acordo com sua afinidade com cada componente de um processo, onde quanto menor a classificação, melhor a combinação do componente com o recurso. O escalonador coloca essas informações em uma *matriz de desempenho*, onde um elemento $e_{i,j}$ da matriz mostra a afinidade do componente i do processo com o recurso j. Finalmente, três heurísticas são executadas sobre a matriz para determinar o escalonamento dos componentes. Apesar do algoritmo não considerar questões relativas às grades computacionais, ele foi desenvolvido para atuar em conjunto com o projeto GrADS, que trata tais problemas. Se utilizado no middleware Xavantes, com controladores, esse algoritmo pode causar a dispersão das tarefas internas a um controlador, resultando em um aumento significativo na comunicação, conseqüentemente tornando o escalonamento ineficiente.

Composição de serviços em grades e escalonamento direcionado a workflow são discutidos e algoritmos são propostos em [29]. Um algoritmo de escalonamento de serviços compostos em grades é proposto. Os serviços de grades compostos são modelados como um workflow, para então serem escalonados na grade. A partir do workflow, um algoritmo de escalonamento baseado em *list scheduling* é utilizado para escalonar os serviços. A técnica de escalonamento apresentada foi desenvolvida para escalonamento de workflows, porém direcionada aos problemas relativos a serviços compostos em grades, não sendo ideal para a aplicação em escalonamento de tarefas.

Em [27] é apresentada uma ferramenta para auxiliar o projeto e avaliação de políticas de escalonamento adequados à execução de aplicações paralelas em grades computacionais.

Um algoritmo que combina escalonamento de tarefas e qualidade de serviço (*quality of service - QoS*) é proposto em [18]. Com a adaptação de técnicas conhecidas de escalonamento de tarefas independentes, o algoritmo introduz QoS e considera recursos não dedicados em grades computacionais. Apenas QoS de uma dimensão é discutida nesse

trabalho, além do algoritmo não tratar escalonamento de tarefas dependentes.

Em [12], um algoritmo de aproximação para escalonamento de tarefas em grades é apresentado, utilizando como critério de desempenho e aproximação o consumo de recursos na grade. O algoritmo assume que todas as tarefas do grafo de dependências têm o mesmo peso, não sendo aplicável a DAGs compostos por tarefas heterogêneas.

DAG Manager (DAGMan) [11] é o gerenciador de execução de DAGs do Condor [22], um conhecido gerenciador de recursos entre domínios utilizado em grades [25]. Dois trabalhos futuros citados no trabalho DAGMan são o suporte a DAGs condicionais e o suporte a DAGs dinâmicos. DAGs condicionais determinam se uma tarefa será executada ou não dependendo de condições lógicas e resultados de outras tarefas. DAGs dinâmicos mudam de acordo com resultados dos nós executados, permitindo aumentar a complexidade dos processos suportados.

Os trabalhos apresentados nesta seção, por não terem sido desenvolvidos para o middleware Xavantes, não consideram a utilização de entidades chamadas *controladores*. Essas entidades, que são utilizadas no middleware Xavantes para o controle de execução das tarefas componentes de um processo, são apresentadas no capítulo seguinte. A distribuição de tarefas resultante do escalonamento realizado por um algoritmo que não considera controladores pode tornar a execução de um processo muito custosa. A estratégia utilizada no algoritmo proposto busca balancear a distribuição de tarefas, tornando o escalonamento resultante compatível com a utilização dos controladores, proporcionando bom desempenho, além das características de recuperação, confiabilidade e tolerância a falhas promovidas pelos controladores.

Capítulo 4

O Middleware Xavantes

O Xavantes foi desenvolvido especificamente para a execução de workflows [3], diferentemente da grande maioria dos middlewares para grades. Estes foram desenvolvidos visando a execução de sacos de tarefas e, quando existente, o suporte a execução de tarefas acopladas depende de extensões desenvolvidas posteriormente, muitas vezes sem suporte completo para workflows. O Xavantes é composto por um modelo de programação, para especificação de processos estruturados, e por uma infra-estrutura, que gerencia tais processos em um ambiente de grades computacionais composto por grupos de recursos altamente distribuídos, heterogêneos e autônomos.

4.1 Modelo de Programação

O modelo de programação do Xavantes permite a especificação de aplicações como processos estruturados, contendo uma hierarquia de estruturas de controle para determinar o fluxo de controle. Um processo pode ser composto por outros processos, controladores e atividades, chamados de elementos de processo. Uma atividade é uma tarefa atômica de um processo, a ser executada em uma única máquina. Um controlador é uma estrutura de controle que organiza a ordem de execução de elementos de processo interiores a ele. Controladores podem ser aninhados, permitindo especificação hierárquica de controle, similarmente às linguagens de programação estruturadas.

Existem vários tipos de controladores, sendo estes classificados como paralelos ou seqüenciais, quanto ao paralelismo, e como simples, condicionais ou interativos, quanto ao controle de fluxo. Os tipos de controladores seqüenciais são similares às estruturas de controle das linguagens de programação estruturadas: *block*, *switch*, *for*, e *while*. Os tipos paralelos equivalem aos seus respectivos tipos seqüenciais, mas executam seus elementos internos simultaneamente: *par*, *parswith*, *parfor* e *parwhile*. Quando aninhados eles podem representar controles mais expressivos, como execução paralela de várias seqüências de

atividades. Ainda, nos controladores paralelos é possível especificar a condição de junção de fluxos (*join*).

A partir do modelo de programação é possível transformar o processo em um DAG. Controladores podem conter apenas atividades, ou atividades e controladores. As Figuras 4.1, 4.2 e 4.3 mostram como cada tipo de controlador é traduzido para a representação em grafos. Cada controlador é representado por um retângulo tracejado, tendo suas tarefas subordinadas em seu interior.

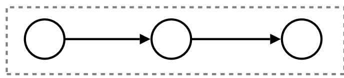

Figura 4.1: Controlador seqüencial.

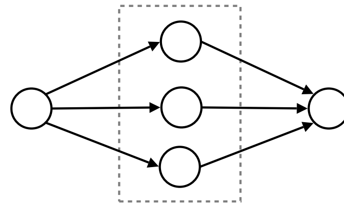

Figura 4.2: Controladores *par* e *parswitch*.

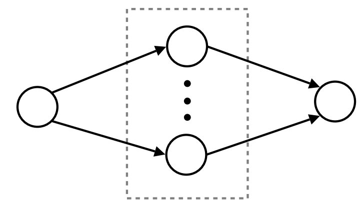

Figura 4.3: Controladores *parfor* e *parwhile*.

A Figura 4.1 mostra a representação de um controlador seqüencial do processo. Nos controladores seqüenciais, cada tarefa é executada após o término da anterior. A Figura 4.2 mostra a representação dos controladores *par* e *parswitch*, que são controladores paralelos e cada uma de suas tarefas pode ser executada paralelamente em um recurso diferente. Esses controladores têm o número de atividades definido no modelo de programação, porém a execução de tais atividades pode depender da condição de execução ou da condição de *join*. No controlador *par*, se a condição de *join* for satisfeita antes do término de todas as tarefas, aquelas que ainda não terminaram sua execução podem ser descartadas. No controlador *parswitch* tarefas são executadas se certa condição de controle é satisfeita. Então, se a condição de execução de uma tarefa não for satisfeita, esta pode ser descartada. Note que essas condições podem depender de resultados obtidos na própria execução das tarefas internas ao controlador.

A Figura 4.3 mostra a representação dos controladores *parfor* e *parwhile*. Nesses controladores o número de atividades depende da condição de entrada. Assim, ao contrário dos controladores *par* e *parswitch*, o número de atividades que podem ser executadas por esses controladores pode não ser conhecida em tempo de compilação, e sim apenas durante a execução do processo.

Além de conter tarefas, cada controlador pode conter outros controladores, criando um aninhamento de controladores. Controladores que contêm controladores em seu interior podem enviar tais controladores para serem executados em outros recursos. Se o controlador pai é um controlador paralelo, os controladores imediatamente subordinados a ele podem ser executados paralelamente.

A utilização de controladores promove a distribuição do controle e da gerência de controladores e tarefas componentes de um processo. Dessa forma, cada controlador tem todas as informações necessárias à execução dos controladores e das tarefas subordinadas a ele. Essa descentralização de controle torna a execução de um processo mais confiável e tolerante a falhas, além de facilitar a recuperação de partes do processo que porventura venham a ser perdidas em alguma falha de recurso. Além disso, os controladores permitem uma especificação de processos inteligível e expressiva.

Um processo completo é representado na Figura 4.4. O retângulo 1 representa um controlador paralelo *par*, contendo as atividades (ou tarefas) 7 e 8. O retângulo 4 representa um controlador seqüencial *block*, contendo as atividades 2, 5, 10 e o controlador *par* 1. O controlador 2 é um controlador seqüencial que contém a atividade 3, enquanto o controlador seqüencial 3 possui as atividades 4, 6 e 9. O controlador 5 é um controlador paralelo *par* que contém os controladores 2, 3 e 4. O controlador 6 é um controlador seqüencial que contém o controlador 5, além do nó de entrada 1 e o nó de saída 11.

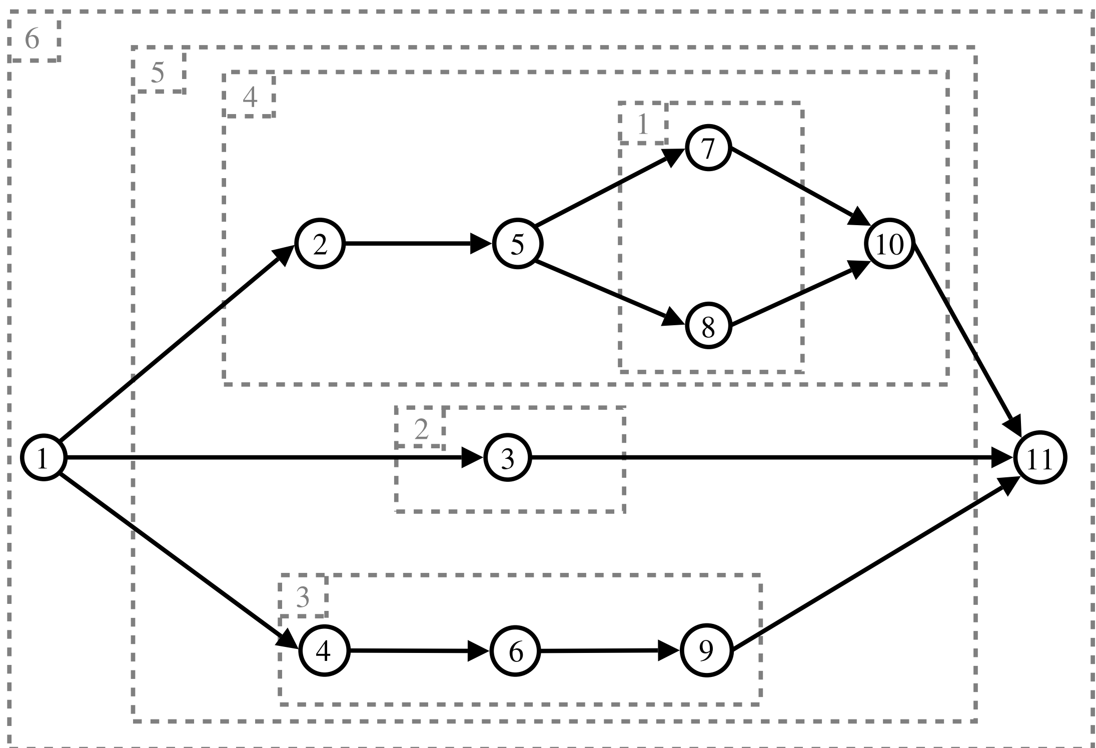

Figura 4.4: Exemplo de DAG representando um processo com controladores.

No Xavantes, os rótulos dos nós e arestas representam número de instruções e bytes transmitidos, respectivamente. Tais valores são obtidos do modelo de programação e são utilizados pelo escalonador para determinar em qual recurso as tarefas serão executadas.

Tendo esses valores e as capacidades de processamento e transmissão dos recursos, o escalonador pode estimar o tempo de execução das tarefas e do processo.

O Xavantes e seu modelo de programação oferecem suporte a grafos condicionais e dinâmicos. Essas duas funcionalidades ainda não são tratadas a fundo pelo algoritmo proposto. Atualmente, nós em grafos condicionais cujas condições de execução são falsas podem ser simplesmente descartados, enquanto nós criados em grafos dinâmicos após o início da execução do processo podem ser escalonados trivialmente na melhor máquina disponível no momento.

4.2 Infra-estrutura

O middleware Xavantes organiza os recursos da grade em grupos análogos às organizações virtuais [10], considerando que máquinas e repositórios de dados que estão próximos, tais como recursos em redes locais e *clusters* de recursos, compõem cada grupo. O principal objetivo desse middleware é oferecer alto desempenho e confiabilidade na execução de processos. Ele gerencia processos estruturados em um ambiente de grade escalonando, distribuindo e executando seus subprocessos, controladores e atividades hierarquicamente nos recursos disponíveis, em um ou mais grupos, de acordo com a estrutura aninhada desses elementos de processo.

Em cada grupo há um *Group Manager* (GM), um ou mais *Process Managers* (PM) e um ou mais *Activity Managers* (AM). O GM é responsável por manter informações sobre a disponibilidade de recursos dentro do grupo, como capacidade de processamento atual, largura de banda e tempo estimado de fila de execução. Além disso, o GM mantém referências aos GMs adjacentes e informações sobre disponibilidade de parte dos recursos desses grupos. Em cada grupo existem um ou mais PMs, que são responsáveis por instanciar e executar os processos, escalonar os elementos de processo e enviá-los para outros recursos. Se um processo contém subprocessos, estes podem ser enviados a outros PMs ou podem ser executados pelo próprio PM que está gerenciando a execução do processo pai. As atividades são enviadas para serem executadas nos recursos (AMs) escolhidos pelo escalonador, que também são responsáveis por manter informações sobre as atividades atribuídas a ele e informar o GM do grupo sobre sua disponibilidade. A Figura 4.5 mostra como os recursos são organizados pelo Xavantes.

Cada recurso que pode ser utilizado para processamento de tarefas tem um AM associado. Quando um recurso ingressa na grade, o AM associado ao recurso envia ao GM do grupo informações sobre as características do recurso.

Para que um processo possa ser executado potencialmente em qualquer recurso da grade, os grupos são conectados entre si, direta ou indiretamente, através dos GMs. Cada GM mantém informações da disponibilidade de parte dos recursos dos grupos adjacentes,

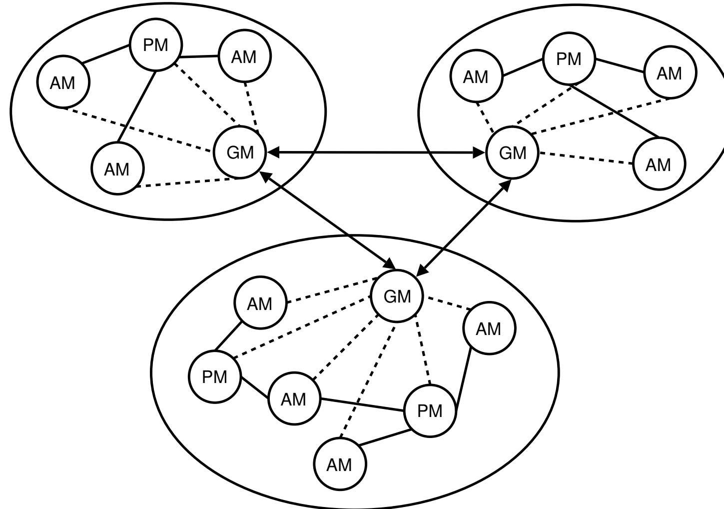

Figura 4.5: Organização dos recursos no middleware Xavantes.

para que o escalonador possa decidir se é vantajoso executar um processo ou tarefa em recursos de outro grupo.

Segundo [3], a execução hierárquica de processos estruturados em vários grupos autônomos e distribuídos de recursos ampliam a escalabilidade da grade. O suporte explícito à execução de processos permite mais eficiência no fornecimento de tolerância a falhas e no escalonamento de processos, melhorando a confiabilidade e o desempenho.

A Figura 4.6 mostra as interações necessárias entre um AM, o GM do grupo e um PM do grupo para a execução de uma tarefa, desde a entrada do AM na grade até o término da execução da tarefa. Inicialmente, o AM envia ao GM sua capacidade de processamento e largura de banda, para que o GM atualize as informações e inclua o novo AM no repositório (interação 1). A partir desse momento, o novo AM está apto a receber requisições de alocação de recursos e execução de tarefas. O PM, que é responsável pelo envio de processos para execução na grade, solicita informações sobre os recursos disponíveis (interação 2). O GM responde à solicitação, buscando as informações no repositório e enviando-as ao PM (interação 3). Com os dados sobre os recursos, o PM pode iniciar o processo de escalonamento das tarefas componentes do processo. Ao final do escalonamento, o PM solicita ao GM a alocação dos recursos necessários à execução do processo (interação 4). O GM repassa o pedido de alocação para cada um dos AMs envolvidos na solicitação (interação 5). Confirmada a alocação e disponibilidade do recurso pelo AM (interação 6), o GM notifica o PM (interação 7), disponibilizando o AM para o PM. O PM envia os dados necessários à execução da tarefa ao AM (interação 8). A tarefa aguarda na fila de tarefas a serem executadas, e ao ser finalizada o AM notifica o PM e envia os dados resultantes da computação (interação 9). Finalmente, o PM notifica o GM

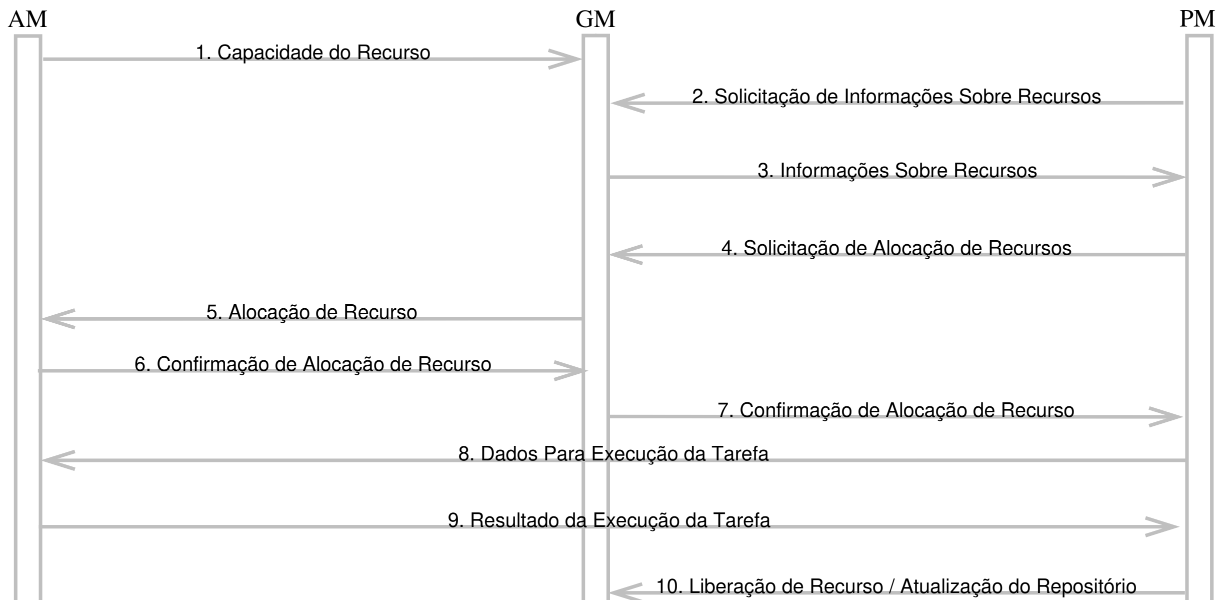

Figura 4.6: Interações entre AM, GM e PM para execução de uma tarefa.

sobre o término da execução da tarefa, para que o repositório de recursos seja atualizado (interação 10).

As interações mostradas na Figura 4.6 são para o caso em que todas as tarefas de um processo são alocadas no mesmo grupo do PM que as está escalonando. O caso em que alguma tarefa é enviada para execução em outro grupo é mostrado na Figura 4.7, onde *AM1* e *GM1* pertencem a um grupo e *GM2* e *PM2* a outro. Nessa figura, o *PM2* solicita um recurso ao *GM2* e, ao receber as informações do repositório e escalonar o processo, decide enviar um *cluster* de tarefas para execução no *AM1*, pertencente a outro grupo.

Nas interações para execução de uma tarefa em grupo adjacente, os passos 1, 2 e 3 são análogos àqueles executados nas interações para execução de tarefa no mesmo grupo. Nas informações enviadas ao PM2 pelo GM2 (interação 3), existem dados sobre recursos disponíveis no grupo 1. Caso o escalonador do PM2 decida executar uma tarefa em um recurso do grupo 1, o PM2 solicita diretamente ao GM1 a alocação de tal recurso (interação 4). Essa requisição é repassada ao AM1 (interação 5), que confirma ao GM1 a alocação (interação 6). O GM1 confirma a alocação ao PM2 (interação 7), disponibilizando o AM1 ao PM2. Os dados necessários à execução da tarefa são enviados pelo PM2 ao AM1 (interação 8). Ao terminar a execução, o AM1 envia os resultados da computação ao PM2 (interação 9), que notifica o GM sobre o término da execução para atualização do repositório (interação 10), completando a execução.

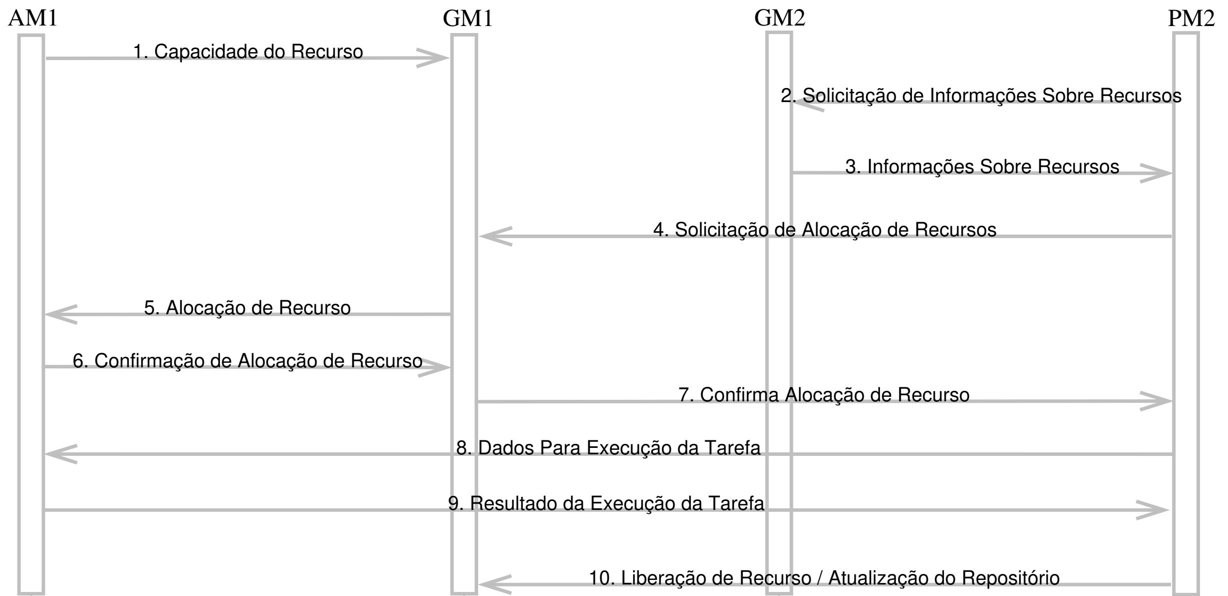

Figura 4.7: Interações entre AM, GMs e PM para execução de uma tarefa em grupo adjacente.

4.3 Exemplo de Processo

Os exemplos do modelo de programação mostrados anteriormente ilustram como é traduzido um controlador do processo especificado no modelo de programação para DAG, porém não têm em sua composição detalhes inerentes ao modelo. O Algoritmo 1 mostra um exemplo completo de um processo programado de acordo com o modelo de programação do Xavantes, enquanto o Algoritmo 2 mostra como uma atividade é programada. Este exemplo contém mais detalhes, explicitando parâmetros de controladores e códigos completos.

O processo exemplificado é uma aplicação de busca de números primos. O processo recebe uma faixa de inteiros dentro da qual deve fazer uma busca por números primos, retornando os encontrados. Para efetuar a computação necessária para decidir se um número é primo ou não, o processo usa um controlador paralelo *PARFOR*, que inicia e despacha várias atividades (tarefas) concorrentes, de mesmo código, de busca de primos.

O processo, definido no Algoritmo 1 e chamado de *FindPrimes*, têm como parâmetros de entrada os limites da faixa de números a ser investigada (linhas 2 e 3), a quantidade de primos a ser devolvida (linha 4) e o número de atividades a serem executadas paralelamente durante a busca (linha 5).

As linhas 8 a 24 são encarregadas de inicializar as variáveis necessárias às atividades de busca de números primos e despachá-las. Entre as linhas 10 e 16 são inicializadas

Algoritmo 1 Processo Xavantes

```
1: <PROCESS_TYPE NAME="FindPrimes">
2:     <IN_PARAMETER TYPE="int" NAME="min"/>
3:     <IN_PARAMETER TYPE="int" NAME="max"/>
4:     <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
5:     <IN_PARAMETER TYPE="int" NAME="numActs"/>
6:     <IN_PARAMETER TYPE="RemoteCollection" NAME="primes"/>
7:     <BODY>
8:     <PARFOR NAME="parfor">
9:         <ITERATE VAR="counter" BEGIN="0" END="numActs - 1">
10:            <CODE>
11:                int range = (max - min + 1) / numActs;
12:                int minNum = range * counter + min;
13:                int maxNum = minNum + range - 1;
14:                if (counter == numActs - 1)
15:                    maxNum = max;
16:            </CODE>
17:            <INVOKE_ACTIVITY TYPE="FindPrimes" NAME="findPrimes">
18:                <IN_PARAMETER NAME="min" VALUE="minNum"/>
19:                <IN_PARAMETER NAME="max" VALUE="maxNum"/>
20:                <IN_PARAMETER NAME="numPrimes" VALUE="numPrimes"/>
21:                <IN_PARAMETER NAME="primes" VALUE="primes"/>
22:                </INVOKE_ACTIVITY>
23:            </ITERATE>
24:        </PARFOR>
25:    </BODY>
26: </PROCESS_TYPE>
```

as variáveis que serão enviadas para cada atividade paralela, onde *numActs* representa o número de atividades a serem utilizadas na busca pelos números primos. A partir do valor de *numActs*, o código determina uma faixa de busca para cada atividade que é despachada, dividindo a faixa total inicial em partes menores. Em cada iteração da linha 9 o código seleciona uma parte de toda a faixa de números e inicia uma atividade paralela, que investiga a faixa recebida.

O Algoritmo 2 mostra a atividade que é utilizada pelo processo. Essa atividade contém o código de busca por números primos e é executada em diferentes recursos, escolhidos pelo escalonador.

A atividade de busca de primos recebe como parâmetros os limites da busca, a quantidade de primos que deve ser encontrada por todo o processo e uma coleção que recebe e guarda os números primos encontrados por todas as atividades (variável *primes*). Deli-

Algoritmo 2 Atividade Xavantes

```
1: <ACTIVITY_TYPE NAME="FindPrimes">
2:     <IN_PARAMETER TYPE="int" NAME="min"/>
3:     <IN_PARAMETER TYPE="int" NAME="max"/>
4:     <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
5:     <IN_PARAMETER TYPE="RemoteCollection" NAME="primes"/>
6:     <CODE>
7:         for (int num = min; num <= max; num++) {
8:             boolean isPrime = true;
9:             int sqrtNum = (int)Math.sqrt(num) + 1;
10:            // verify if num is a prime
11:            for (int i = 2; i < sqrtNum; i++) {
12:                if (num % i == 0) {
13:                    isPrime = false;
14:                    break;
15:                }
16:            }
17:            if (isPrime) {
18:                // stop, required number of primes was found
19:                if (primes.size() >= numPrimes)
20:                    break;
21:                primes.add(new Integer(num));
22:            }
23:        }
24:    </CODE>
25: < /ACTIVITY_TYPE>
```

mitado pelo início e fim do rótulo *CODE*, entre as linhas 6 e 24, está um código simples de busca e identificação de números primos, que itera sobre a faixa de números recebida. Cada primo encontrado é adicionado à coleção *primes* caso o número de primos necessário ainda não tenha sido encontrado. Caso contrário, a atividade termina sua execução.

Uma das vantagens do uso de controladores é a possibilidade do middleware interpretar seu conteúdo e determinar em que ponto a execução do processo está, permitindo recuperação automática e monitoramento. Ainda, devido à natureza estruturada dos controladores, estes podem ser facilmente compostos, e o middleware pode tirar vantagem dessa composição para distribuir hierarquicamente os controladores aninhados em diferentes recursos da grade, mantendo tarefas fortemente acopladas em recursos próximos e distribuindo mais amplamente, se necessário, as fracamente acopladas. Essa característica fornece desempenho, escalabilidade e confiabilidade [3].

Capítulo 5

Escalonamento no Middleware Xavantes

Neste capítulo detalhamos alguns aspectos do middleware que são relevantes ao escalonamento. O Xavantes organiza os recursos da grade em grupos, considerando que máquinas e repositórios de dados que são próximos, tais como recursos em redes locais e *clusters*, compõem cada grupo. Em cada grupo há um *Group Manager* (GM), um ou mais *Process Managers* (PM) e um ou mais *Activity Managers* (AM). O GM é responsável por manter informações de disponibilidade de recursos dentro do grupo, além de referências aos GMs adjacentes e informações sobre alguns recursos dos grupos adjacentes. As informações mantidas pelo GM são utilizadas nas tomadas de decisão do escalonador.

Em cada grupo existem um ou mais PMs, que são responsáveis por instanciar e executar os processos, escalonar os elementos de processo e enviá-los para outros recursos. As atividades são enviadas para serem executadas nos recursos (AMs) escolhidos pelo escalonador. Processos podem ser enviados a outros PMs ou podem ser executados pelo próprio PM.

O processo de escalonamento é realizado nos *Process Managers*, com parte do escalonador localizada nos GMs para obter informações sobre disponibilidade de recursos. Inicialmente o escalonador recebe um workflow como entrada, representado pelo modelo de programação e transformado em um DAG, e requisita recursos ao GM do grupo. Então o algoritmo de escalonamento é executado, analisando o DAG e escalonando as tarefas. Com todas as tarefas atribuídas aos recursos, o PM requisita ao GM do grupo os recursos necessários. O PM então despacha cada atividade para execução sob o controle dos AMs nos recursos selecionados durante o escalonamento. A Figura 5.1 mostra a localização do escalonador na arquitetura.

Na Figura 5.1, na representação do *Group Manager* temos:

- Comunicação entre GM e AMs

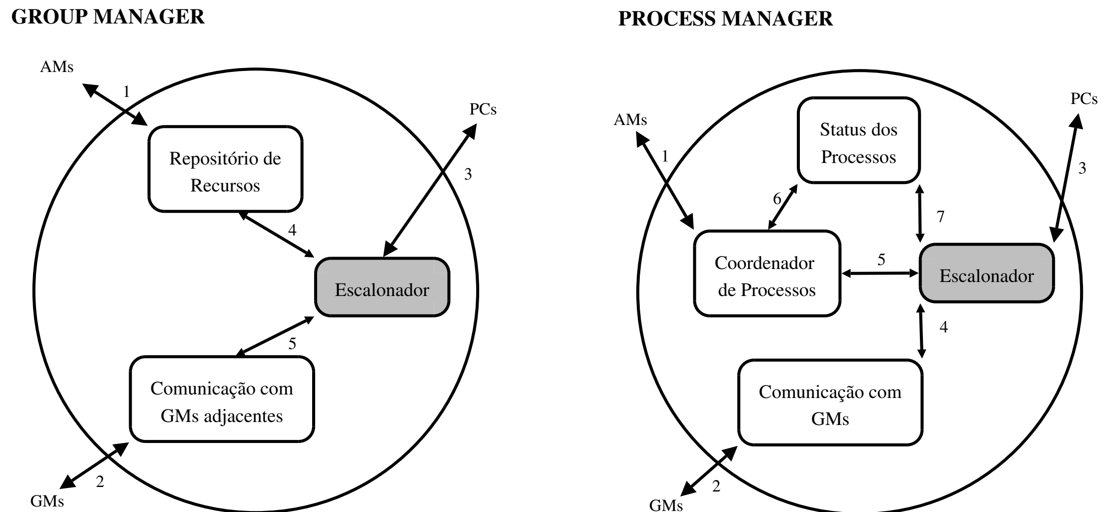

Figura 5.1: Escalonador na arquitetura.

- AMs enviam informações sobre disponibilidade ao GM do grupo.

- GM requisita espaço na fila para execução de tarefas (alocação de recurso).

- AM confirma alocação de recurso ao GM.

- Comunicação entre GMs

  - GM envia informações sobre disponibilidade de parte de seus recursos aos GMs adjacentes.

- Comunicação entre GMs e PMs

  - PMs requisitam informações sobre disponibilidade de recursos.

  - GM envia ao PM dados sobre recursos disponíveis.

  - PM requisita alocação de recursos a um GM, local ou de outro grupo.

  - GM confirma alocação de recurso ao PM.

  - PMs notificam GM sobre término de execução de processos e conseqüente liberação de recursos.

- Comunicação interna do GM

  - Escalonador busca recursos no repositório.

Na representação do *Process Manager*, além da comunicação entre GMs e PMs análoga à descrita na representação do *Group Manager*, temos:

- Comunicação entre PM e AMs
  - PM envia tarefas para execução nos AMs.
  - PMs enviam dados e determinam o início da execução de tarefas.
  - Tarefas em execução nos AMs enviam dados para seu controlador no PM.
- Comunicação entre PMs:
  - PM envia componente de processo para ser executado em outro PM.
- Comunicação interna do PM:
  - Escalonador requisita recurso ao GM.
  - Escalonador envia atividade para execução.
  - Controle de estado dos processos.
  - Escalonador verifica estado dos processos para controle do workflow.

A atribuição de tarefas é de responsabilidade dos *Process Managers*, que as despacham a um *Activity Manager* do grupo. Um PM pode atribuir a execução de uma tarefa a um AM de outro grupo. Para isso, o PM faz a requisição do recurso ao GM de outro grupo, que confirma a disponibilidade do recurso. Então o PM envia a atividade para execução no AM do outro grupo. Os processos são submetidos como workflows especificados pelo modelo de programação.

Os PMs do grupo são responsáveis por receber os processos e enviá-los ao escalonador. Em seguida, o escalonador obtém as informações sobre disponibilidade de recursos na grade, do seu e de outros grupos, através do GM do seu grupo. Finalmente, o PM envia as tarefas e controladores para serem executados nos AMs, de acordo com o escalonamento retornado pelo algoritmo aqui proposto. A alocação de recursos é requisitada ao GM do grupo, caso os recursos estejam no mesmo grupo, ou a um GM externo, caso os recursos estejam em outro grupo.

Recebido pelo escalonador um processo especificado pelo modelo de programação, o primeiro passo para escaloná-lo é transformá-lo em um DAG. Nos grafos que podem ser especificados pelo modelo de programação do Xavantes cada *fork* tem um *join* correspondente. Um exemplo de grafo é mostrado na Figura 5.2, onde cada *fork* tem seu *join* correspondente. Ainda, *forks* e *joins* devem ser aninhados, isto é, um *fork* deve sempre ter seu *join* correspondente interno ao *fork-join* pai. Essa é uma restrição do modelo de programação análoga à restrição encontrada nas linguagens de programação estruturadas, onde um comando *for* aninhado a um comando externo deve iniciar e terminar dentro desse comando externo. Essa restrição pode ser superada utilizando variáveis compartilhadas dentro dos controladores na especificação do processo, analogamente ao uso de variáveis globais nas linguagens de programação estruturadas.

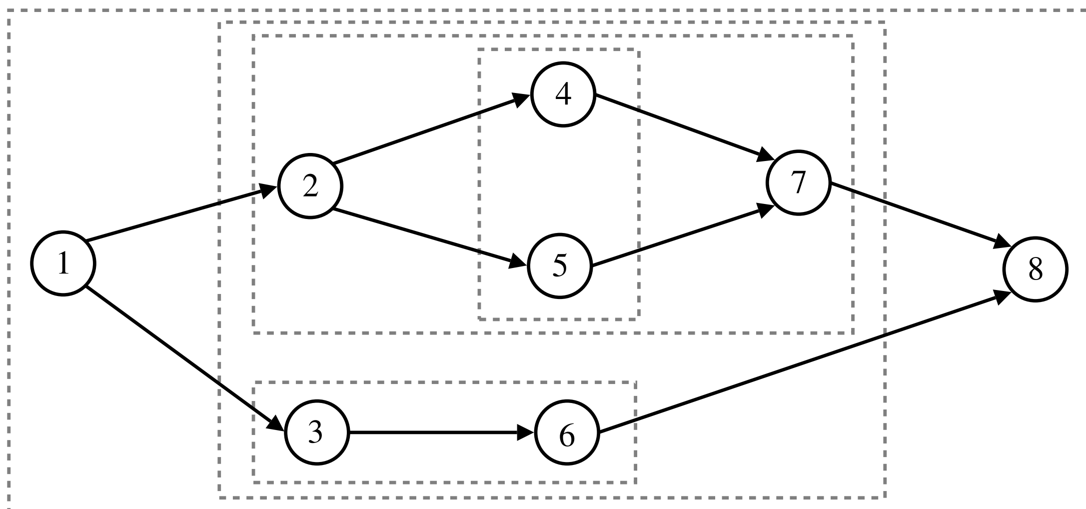

Figura 5.2: Grafo aceito pelo modelo de programação.

O Algoritmo 3 mostra uma possível representação do DAG da Figura 5.2 no modelo de programação. O controlador *BLOCK* da linha 1 engloba diretamente as tarefas 1 e 8, além de ser o superior direto do controlador *PAR* da linha 3. Este último não possui nenhuma tarefa subordinada diretamente a ele, sendo responsável pelo controle da execução dos controladores *BLOCK*, linha 4, e *BLOCK*, linha 12. Note que, por exemplo, a comunicação entre os nós 1 e 2 deve passar pelos controladores *PAR* e *BLOCK* das linhas 3 e 4. Seguindo a interpretação do código, o controlador *PAR* da linha 6 contém as tarefas 4 e 5, não havendo nenhum controlador subordinado a ele, com o controlador sendo finalizado na linha 9. Os dados resultantes das computações das tarefas 4 e 5 são enviados ao controlador *BLOCK* pai, para que sejam utilizados pela tarefa 7. Após a especificação da tarefa 7 no código, o controlador *BLOCK* que contém as tarefas 3 e 6 é codificado, com início na linha 12 e fim na linha 15. O controlador *BLOCK* que inicia na linha 3 é finalizado na linha 16. Os resultados das tarefas 6 e 7 são enviados para a tarefa 8, passando pelo controlador finalizado na linha 16. Finalmente, a tarefa 8 é especificada na linha 17 e o código é finalizado com o término do controlador *BLOCK* mais externo.

Na Figura 5.3 existem dependências de dados entre tarefas que estão em controladores diferentes e esses controladores estão diretamente subordinados a um mesmo controlador, o que implicaria em uma intersecção de dois controladores no modelo de programação. Dessa forma, tal grafo não poderia ser representado pelo modelo de programação do Xavantes utilizando apenas os controladores.

Uma representação do grafo da Figura 5.3 no modelo de programação seria como mostra o Algoritmo 4. Observe que a estrutura hierárquica do código não representa as dependências de dados entre as tarefas 4 e 8 e entre as tarefas 6 e 7, pois não é possível

Algoritmo 3 Código para Figura 5.2

```
1: <BLOCK>
2:     Tarefa 1
3:     <PAR>
4:         <BLOCK>
5:             Tarefa 2
6:             <PAR>
7:                 Tarefa 4
8:                 Tarefa 5
9:             </PAR>
10:            Tarefa 7
11:        </BLOCK>
12:        <BLOCK>
13:            Tarefa 3
14:            Tarefa 6
15:        </BLOCK>
16:    </PAR>
17:    Tarefa 8
18: </BLOCK>
```

representá-las sem criar um código inconsistente com o modelo de programação. Para sanar esse problema, a dependência deve ser programada de forma diferente, criando uma variável compartilhada dentro dos controladores para que os dados da dependência sejam acessados por ambas as tarefas que definem cada dependência. Essas variáveis compartilhadas são mostradas no código nas linhas 7 e 14.

O modelo de programação fornece para o escalonador o número de instruções de cada tarefa e a quantidade de dados que será transmitida entre as tarefas (dependência de dados). Para determinar a capacidade dos recursos, *benchmarks* são executados. Durante a execução de um processo, as tarefas estão subordinadas a controladores, que são os elementos de controle responsáveis pela execução das tarefas. Controladores são elementos especificados no modelo de programação que permitem alta distribuição dos processos. O *escalonamento de controladores* é o passo do algoritmo proposto que visa minimizar o *overhead* de comunicação gerado pelos controladores. O *overhead* gerado por esses elementos é justificado pelas vantagens oferecidas por eles, como alta disponibilidade, escalabilidade, tolerância a falhas e recuperação, confiabilidade e desempenho. Qualquer comunicação entre duas tarefas é feita através de seus controladores. Maiores detalhes sobre o Xavantes estão em [3].

A Figura 5.4 mostra uma representação de um processo e seus controladores. Rótulos dos nós são números de instruções de cada tarefa (custos de computação) e rótulos das

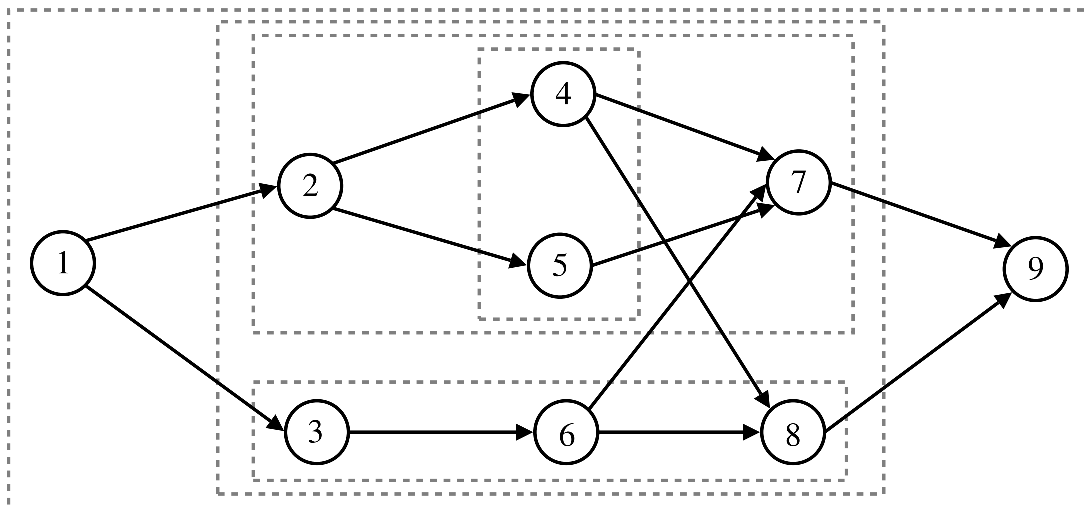

Figura 5.3: Grafo não aceito pelo modelo de programação.

arestas são dados transmitidos (custos de comunicação). Os controladores são representados pelos retângulos e são obtidos do modelo de programação. Nesse exemplo, a tarefa 1 é a primeira a ser executada, tendo as tarefas 2, 3 e 4 como dependentes. Estas só podem iniciar as suas execuções após receberem os dados computados pela tarefa 1. As outras dependências podem se comportar de forma análoga ou podem estar sujeitas a condições, tornando o grafo condicional e/ou dinâmico. Em relação aos controladores, as tarefas 7 e 8 estão subordinadas ao controlador 1, a tarefa 3 ao controlador 2, as tarefas 4, 6 e 9 ao controlador 3 e assim por diante. Note que, por exemplo, a comunicação entre os nós 6 e 9 deve passar pelo controlador 3. Então, supondo que esses dois nós executam no recurso r_a e o controlador 3 executa no recurso r_b, para o que o nó 6 envie os dados ao nó 9, o nó 6 deve enviar os dados para o controlador 3, no recurso r_b, e este deve enviar os dados ao nó 9, no recurso r_a. Nesse caso a comunicação entre os nós 6 e 9, ambos executando no recurso r_a, não teria custo zero. O *escalonamento de controladores* promovido pelo algoritmo proposto tem como objetivo minimizar comunicações desse tipo, evitando assim um *overhead* alto de comunicação. O escalonamento de controladores é feito com todas as tarefas já escalonadas. Observe que tarefas subordinadas a controladores seqüenciais formam caminhos no grafo de dependências, fato que foi considerado ao desenvolver o algoritmo de agrupamento de tarefas. Agrupando-se tarefas seqüenciais do grafo e escalonando-as no mesmo recurso que o controlador ao qual elas estão subordinadas, a comunicação entre tarefas e controladores é suprimida, eliminando parte do *overhead* de comunicação gerado pelos controladores.

Algoritmo 4 Código para Figura 5.3

```
1: <BLOCK>
2:     Tarefa 1
3:     <PAR>
4:         <BLOCK>
5:         Tarefa 2
6:             <PAR>
7:                 Variável compartilhada com Tarefa 8
8:                 Tarefa 4
9:                 Tarefa 5
10:            </PAR>
11:            Tarefa 7
12:        </BLOCK>
13:        <BLOCK>
14:            Variável compartilhada com Tarefa 7
15:            Tarefa 3
16:            Tarefa 6
17:            Tarefa 8
18:        </BLOCK>
19:    </PAR>
20:    Tarefa 9
21: </BLOCK>
```

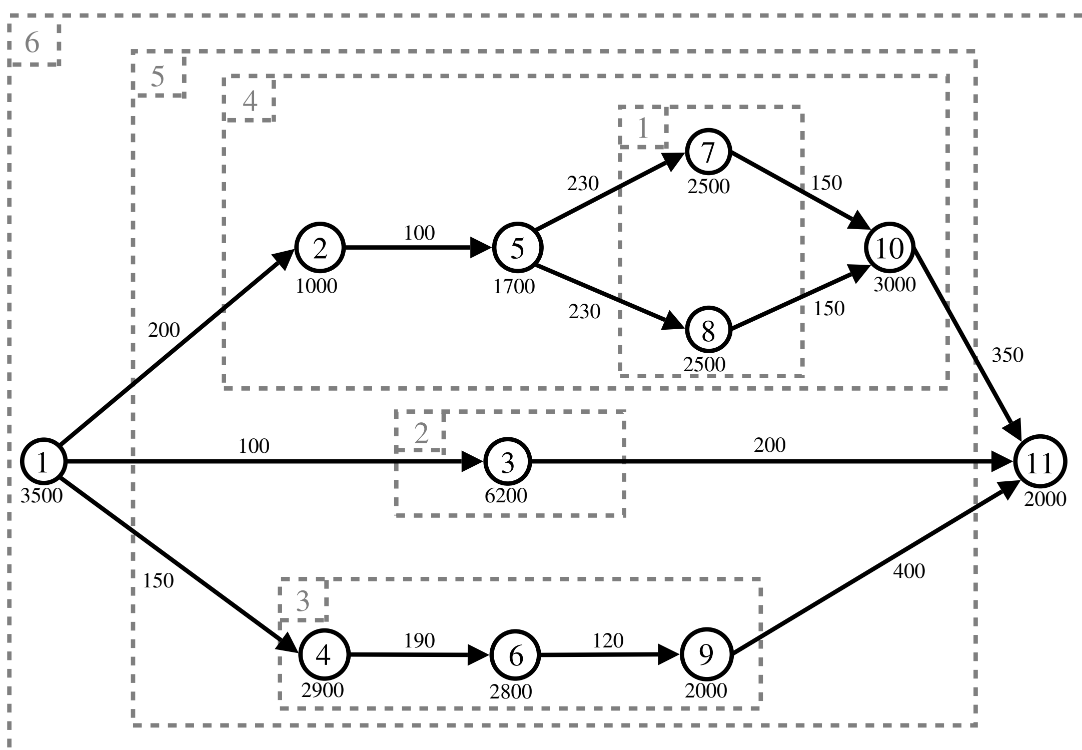

Figura 5.4: Representação completa de um processo.

Capítulo 6

Algoritmo Proposto

Utilizando técnicas de escalonamento em sistemas heterogêneos e tendo o middleware Xavantes [3] como base, um algoritmo de escalonamento foi desenvolvido [1]. O algoritmo *Path Clustering Heuristic* (PCH) considera a heterogeneidade do sistema e a existência de controladores para promover o escalonamento de tarefas, com recuperação e tolerância a falhas fornecidos pelo middleware Xavantes. As aplicações alvo do PCH são aquelas que podem ser representadas pelo modelo de programação Xavantes em um cenário composto por grupos de recursos heterogêneos altamente distribuídos.

O PCH é baseado na técnica de *clustering*. Chamamos de *cluster* um agrupamento de tarefas que o algoritmo constrói. As tarefas que fazem parte de um mesmo *cluster* são escalonadas em um mesmo recurso.

Em uma visão geral, na criação de um *cluster* o algoritmo PCH seleciona um caminho do DAG, fazendo uma busca em profundidade, e escalona seus nós no mesmo processador, com o objetivo de eliminar comunicação entre nós que têm dependência de dados e entre nós e controladores. Com todos os nós escalonados, o algoritmo inicia o escalonamento de controladores, necessário para minimizar a comunicação entre controladores e tarefas. Essa visão geral é mostrada no Algoritmo 5.

Algoritmo 5 PCH - visão geral

1: **while** existem nós não escalonados **do**
2: $cluster \Leftarrow$ gera_próximo_agrupamento()
3: $recurso \Leftarrow$ seleciona_melhor_recurso($cluster$)
4: Escalona $cluster$ em $recurso$
5: **end while**
6: escalona_controladores()

A política de criação de *clusters* com tarefas pertencentes a caminhos do grafo tem como objetivo minimizar o *overhead* de comunicação gerado pelos controladores. As

tarefas que estão no mesmo *cluster*, ou agrupamento, são executadas no mesmo recurso. Com essa política de agrupamento de caminhos, tarefas que pertencem a um controlador seqüencial sem subcontroladores estarão no mesmo agrupamento, e conseqüentemente serão executadas no mesmo recurso. Dessa maneira, executando o controlador no mesmo recurso de suas tarefas, a comunicação é minimizada e o *overhead* dos controladores pode ser reduzido a zero. Como toda comunicação entre tarefas deve passar por seus respectivos controladores, se as tarefas pertencentes a um controlador seqüencial estivessem dispersas em vários recursos, a comunicação entre tarefas poderia sofrer um acréscimo muito grande, tornando o desempenho global sofrível.

6.1 Definições

Alguns atributos são calculados para as tarefas componentes do processo. Esses atributos são utilizados pelo algoritmo de escalonamento para tomar decisões. Os atributos são calculados para cada tarefa do grafo e são comuns a vários algoritmos de escalonamento de tarefas. Esses algoritmos se diferenciam no método utilizado para selecionar as tarefas a serem escalonadas e os processadores que irão executar tais tarefas. Os atributos utilizados no PCH refletem a prioridade e o custo para executar cada tarefa do processo, e são definidos como:

- *Weight (Custo de Computação)*

$$w_i = \frac{instruções_i}{processamento_r}$$

Representa o custo de computação (tempo de execução) do nó i no recurso r, onde $processamento_r$ é o poder de processamento do recurso r, em instruções por segundo.

- *Custo de Comunicação*

$$c_{i,j} = \frac{dados_{i,j}}{banda_{r,t}}$$

Representa o custo de comunicação (tempo de transmissão dos dados) entre os nós i e j, usando o link entre os recursos r e t. Se $r = t$, $c_{i,j} = 0$.

- *Prioridade*

$$P_i = w_i + \max_{n_j \in suc(n_i)} (c_{i,j} + P_j)$$

Representa o nível de prioridade do nó n_i. A prioridade do nó de saída é $P_{\text{saída}} = w_{\text{saída}}$.

- *Earliest Start Time*

$$EST(n_i, r_k) = \max\{tempo(r_k), \max_{n_h \in pred(n_i)} (EST_h + w_h + c_{h,i})\}$$

Representa o menor tempo inicial possível para o nó n_i no recurso r_k. Nesta definição, $tempo(r_k)$ representa o tempo em que o recurso r_k estará disponível para executar a tarefa n_i. O EST da primeira tarefa do processo é o tempo em que o processador que contém $n_{entrada}$ estará pronto para executá-la.

- *Estimated Finish Time*

$$EFT(n_i, r_k) = EST(n_i, r_k) + \frac{instruções_{n_i}}{processamento_{r_k}}$$

Representa o tempo estimado de término do nó n_i no recurso r_k.

Os valores iniciais dos atributos citados acima são calculados supondo um sistema homogêneo virtual composto por um número ilimitado de processadores. Esses processadores têm o poder computacional do melhor processador disponível no sistema heterogêneo real, e todos os links têm a largura de banda mais alta disponível no sistema heterogêneo real. Cada tarefa é escalonada em um processador diferente nesse sistema virtual, então o algoritmo computa os valores dos atributos de cada nó.

6.2 Seleção de Tarefas e Agrupamento

Na seleção de tarefas e agrupamento (*clustering*), o algoritmo escolhe as tarefas que formarão cada *cluster*. O recurso que comportará cada *cluster* é escolhido de acordo com a estratégia de seleção de recursos apresentada na seção 6.3. A estratégia adotada no agrupamento de tarefas objetiva reduzir a comunicação entre tarefas acopladas e entre tarefas e controladores. A primeira tarefa que compõe um *cluster* é selecionada de acordo com um critério de prioridade. No PCH tal critério é o valor do atributo *Prioridade* de cada tarefa. A partir dessa tarefa, o algoritmo faz uma busca em profundidade, selecionando novas tarefas para serem adicionadas ao *cluster*. O *cluster* é finalizado quando atinge uma tarefa que obedece ao critério de parada da busca em profundidade. Note que com a busca em profundidade, o algoritmo caminha sobre tarefas que possivelmente estão em um mesmo controlador, adicionando-as a um mesmo *cluster*. Dessa forma, tarefas subordinadas a um mesmo controlador são escalonadas no mesmo processador, minimizando a comunicação entre elas. Como várias tarefas pertencentes a um mesmo controlador estão no mesmo processador, a execução de tal controlador nesse processador contribuirá para a diminuição do *overhead* de comunicação durante execução do processo.

6.2.1 Algoritmo de Seleção de Tarefas e Agrupamento

Computados os valores iniciais dos atributos é possível utilizar a prioridade dos nós para selecionar as tarefas. O primeiro nó selecionado para compor um *cluster* cls_k é o nó n_i não escalonado com maior atributo *Prioridade*. O próximo nó selecionado para compor cls_k é o nó $n_s \in suc(n_i)$ com o maior $P_s + EST_s$. Essa soma representa o tamanho do maior caminho que inicia em $n_{entrada}$, passa por n_s e termina em $n_{\text{saída}}$.

No início do escalonamento o primeiro nó a ser selecionado será o nó de entrada, já que a função recursiva de cálculo de prioridade acumula os valores dos sucessores, ficando o nó de entrada com a maior prioridade. Após o nó de entrada ser selecionado, ele é adicionado ao *cluster* cls_0. Agora o algoritmo realiza uma busca em profundidade, partindo do nó de entrada, selecionando sempre o sucessor n_s que tem maior $P_s + EST_s$ e adicionando-o ao *cluster* cls_0. A busca em profundidade continua até que o nó de saída seja alcançado, que também é adicionado ao *cluster* cls_0.

Após o escalonamento do primeiro *cluster*, que contém o caminho crítico do grafo inicial, a supressão da comunicação entre os nós que compõem esse *cluster* pode fazer com que um outro caminho se torne o caminho crítico no novo grafo. Por esse motivo, a cada *cluster* escalonado o algoritmo atribui o valor 0 às arestas existentes entre tarefas no mesmo recurso e recalcula o Weight, o EST e o EFT de cada tarefa no grafo.

Para formar o próximo *cluster*, o algoritmo seleciona o nó n_i não escalonado com maior prioridade, adicionando-o ao *cluster* cls_k. Partindo desse nó o algoritmo efetua uma busca em profundidade de forma análoga à realizada durante a formação do primeiro *cluster*, porém a busca cessa quando atinge um nó sem sucessores não escalonados, incluindo-o no *cluster.* Então o *cluster* é escalonado e os atributos do grafo recalculados. O Algoritmo 6 mostra como é realizado o agrupamento de tarefas.

Algoritmo 6 gera_próximo_agrupamento

1: $n \Leftarrow$ nó não escalonado com a maior Prioridade.
2: $cluster \Leftarrow cluster \ \cup \ n$
3: **while** (n tem sucessores não escalonados) **do**
4: $\quad n_{suc} \Leftarrow sucessor_i$ de n com maior $P_i + EST_i$
5: $\quad cluster \Leftarrow cluster \ \cup \ n_{suc}$
6: $\quad n \Leftarrow n_{suc}$
7: **end while**
8: **return** *cluster*

6.2.2 Exemplo de Execução

Considerando os recursos da Tabela 6.1 e o grafo da Figura 5.4, o algoritmo gera os *clusters* mostrados na Tabela 6.2. O primeiro nó a ser adicionado ao *cluster* 0 é o nó 1, que é o nó de entrada. O próximo nó selecionado e adicionado ao *cluster* 0 é o nó 2, já que $P_2 + EST_2 > P_3 + EST_3$ e $P_2 + EST_2 > P_4 + EST_4$. Seguindo o mesmo raciocínio, os próximos nós a serem adicionados ao *cluster* 0 são o nós 5, 7, 10 e 11, que é o nó de saída. Observe que este primeiro *cluster* é composto por todos os nós que compõem o caminho crítico do DAG inicial. O caminho crítico é determinante no tempo de execução do processo, então é importante escaloná-lo em um recurso que proporcione um bom desempenho. Então, o *cluster* 0 é o primeiro a ser escalonado, de acordo com a estratégia de seleção de recursos.

No grafo da Figura 5.4, com o nó 1 já escalonado, o nó 4 é o nó com maior prioridade. O nó 4 é então adicionado ao *cluster* 1 e, de forma trivial, o *cluster* é completado com os nós 6 e 9. Note que, em virtude da estrutura dos grafos que são gerados pelo modelo de programação e do algoritmo de composição dos *clusters*, cada *cluster* tem apenas uma tarefa que o sucede no grafo. Os próximos *clusters* gerados no exemplo são: *cluster* 2, com o nó 8, e o *cluster* 3, com o nó 3.

Tabela 6.1: Recursos usados no exemplo da Figura 5.4. A largura de banda entre todos os recursos é fornecida.

Recurso	Processamento	Banda			
0	133	∞	10	5	5
1	130	10	∞	5	5
2	118	5	5	∞	10
3	90	5	5	10	∞

6.3 Seleção de Recursos

O passo de seleção de recursos é responsável por determinar em qual recurso cada *cluster* de tarefas criado será executado. A escolha do recurso para cada *cluster* é tão importante quanto o passo de agrupamento de tarefas, pois determina o tempo de execução de cada *cluster*, influenciando diretamente no tempo total de execução do processo. A seleção de recursos se dá através do cálculo de valores estimados de início, tempo de execução e término para o *cluster* em cada recurso disponível. O cálculo em cada recurso tenta prever qual recurso terminará a execução do *cluster* em menor tempo. Entretanto, não

Tabela 6.2: Atributos dos nós após executar o algoritmo para o grafo da Figura 5.4.

Nó	Recurso	Prioridade	EST	EFT	w	Cluster
1	0(133)	206.0	0	26.3	26.3	0
2	0(133)	159.7	26.3	33.8	7.5	0
3	2(118)	81.7	46.3	98.8	52.5	3
4	1(130)	143.9	41.3	63.6	22.3	1
5	0(133)	142.2	33.8	46.6	12.8	0
6	1(130)	103.1	63.6	85.1	21.5	1
7	0(133)	106.4	46.6	65.4	18.8	0
8	0(133)	106.4	65.4	84.2	18.8	2
9	1(130)	70.1	85.1	100.5	15.4	1
10	0(133)	72.6	84.2	106.8	22.6	0
11	0(133)	15.0	140.5	155.5	15.0	0

só o tempo de término da execução de um *cluster* é determinante no tempo total de execução do processo. Ao fim da execução, um *cluster* deve enviar seus dados à próxima tarefa do DAG, que é a tarefa sucessora da última tarefa do *cluster* considerado para escalonamento. Um *cluster* pode executar rapidamente em um recurso, mas o envio vagaroso dos dados necessários à execução de outra tarefa pode atrasar demasiadamente o início de sua execução. Portanto, no algoritmo de seleção de recursos do PCH, o que define o recurso que executará um *cluster* é o tempo estimado de início da tarefa dependente do último nó do *cluster* que está sendo escalonado. Sendo assim, o algoritmo primeiro estima o tempo de término de todas as tarefas do *cluster* e em seguida estima o tempo de início da tarefa sucessora, que faz parte de outro *cluster*. O recurso selecionado é aquele que minimiza o tempo de início do nó que sucede o *cluster* no grafo de dependências. No caso do primeiro *cluster*, que não tem sucessor por ter o nó de saída em sua composição, o recurso escolhido é o que minimiza o tempo de término do nó de saída.

6.3.1 Algoritmo de Seleção de Recursos

Na heurística de agrupamento de caminhos proposta na seção anterior cada *cluster* contém um caminho do grafo que será escalonado no mesmo recurso. O fator que determina em qual recurso um *cluster* será escalonado é o EST do sucessor do último nó do *cluster* considerado. Em outras palavras, um *cluster* cls_k é escalonado no recurso que minimiza o EST do sucessor de cls_k. Para calcular esse valor, o algoritmo primeiro calcula o Estimated Finish Time (EFT) de cada nó do *cluster*. É importante salientar que se o recurso contém tarefas do mesmo processo do *cluster* que está sendo escalonado, é necessário antes ordenar

as tarefas para que suas precedências não sejam violadas, e então efetuar o cálculo dos EFTs. É fácil ver que se as tarefas estiverem ordenadas em ordem não crescente de prioridade, então suas precedências são satisfeitas.

Por exemplo, considere um recurso r_n com as tarefas (n_3, n_6, n_{10}) escalonadas, um *cluster* $cls_k = (n_8, n_{12}, n_{13})$, e as prioridades $p_3 > p_6 > p_8 > p_{10} > p_{12} > p_{13}$. Para que seja possível determinar os EFTs dos nós do *cluster* cls_k no recurso r_n, é preciso considerar o escalonamento $(n_3, n_6, n_8, n_{10}, n_{12}, n_{13})$.

Com os EFTs das tarefas determinados, o cálculo do EST do sucessor de cls_k que está sendo escalonado é direto: $EST(suc(n_{k_z})) = EFT(n_{k_z}) + c(n_{k_z}, suc(n_{k_z}))$, onde n_{k_z} é o último nó do *cluster* cls_k. O primeiro *cluster*, cls_0, não possui sucessores, então o recurso escolhido para executar o *cluster* cls_0 é o que propicia menor $EFT(n_{0_z})$. Os outros *clusters* têm somente um sucessor e esse sucessor já está escalonado, pois, por construção, o último nó de um *cluster* não tem sucessores não escalonados. A estratégia de seleção de recursos é mostrada no Algoritmo 7.

Algoritmo 7 seleciona_melhor_recurso

1: **for all** $r \in recursos$ **do**
2:     *escalonamento* $\Leftarrow$ Insere *cluster* em $escalonamento_r$
3:     calcula_EFT(*escalonamento*);
4:     $tempo_r \Leftarrow$ calcula_EST(sucessor(*cluster*))
5: **end for**
6: **return** recurso r com menor $tempo_r$

6.3.2 Exemplo de Execução

A Tabela 6.3 mostra os resultados da execução do algoritmo de seleção de recursos para o exemplo da Figura 5.4. Usando a estratégia de seleção de recursos descrita, o *cluster* 0 é escalonado no recurso 0, pois seu tempo de término nesse recurso é de 103.0, o menor entre os recursos disponíveis. Para os *clusters* subseqüentes, o tempo considerado é o tempo de início do sucessor do *cluster.* Assim, o *cluster* 1 é escalonado no recurso 1, que resulta em tempo de início 140.5 para seu sucessor. O *cluster* 2 é escalonado no recurso 0, com o nó 8 inserido antes do nó 10, obedecendo as precedências das tarefas. O tempo de início do sucessor do *cluster* 2 é estimado em 84.2. Finalmente, o *cluster* 3 é escalonado no recurso 2, com tempo de início para seu sucessor de 128.9.

Tabela 6.3: Resultados da seleção de recursos para o grafo da Figura 5.4.

Cluster	Recurso	Tempo início sucessor
0	0(133)	103.0
0	1(130)	105.4
0	2(118)	116.1
0	3(90)	152.2
1	0(133)	145.9
1	1(130)	140.5
1	2(118)	201.6
1	3(90)	221.9
2	0(133)	84.2
2	1(130)	103.8
2	2(118)	143.8
2	3(90)	150.4
3	0(133)	153.4
3	1(130)	152.9
3	2(118)	138.9
3	3(90)	155.2

6.4 Escalonamento de Controladores

Os controladores desempenham um importante papel na especificação e execução dos processos, então estes devem ser escalonados procurando oferecer respostas imediatas às tarefas subordinadas a eles. Assim, devemos atribuir cada controlador a um processador que minimize a comunicação com seus nós e com seus subcontroladores. A política de criação de *clusters* baseados em tarefas seqüenciais favorece na redução do *overhead* de comunicação dos controladores.

6.4.1 Algoritmo de Escalonamento de Controladores

O controlador selecionado para ser escalonado é uma escolha aleatória entre aqueles que têm todos seus subcontroladores já escalonados. Desta forma, o primeiro controlador a ser escalonado será um dos que não têm subcontroladores. Para decidir em que recurso um controlador irá executar, o algoritmo calcula a *comunicação externa* e a *comunicação interna* do controlador. Na comunicação externa estão incluídas as arestas que ultrapassam os limites do controlador, que na representação gráfica são aquelas que interceptam um dos lados do retângulo do controlador. Na comunicação interna estão incluídas apenas

as arestas que não interceptam nenhum dos lados do retângulo do controlador considerado. Então, para calcular a comunicação de um controlador, os nós são separados em duas classes: nós apenas com comunicação interna e nós com comunicação externa ao controlador.

Para calcular a comunicação de um nó n_i com comunicação interna é suficiente somar os custos de comunicação de n_i com seus predecessores e com seus sucessores. Note que se dois nós n_i e n_j estão subordinados diretamente ao mesmo controlador ctr_k e existe uma aresta entre eles, duas comunicações são necessárias: n_i envia os dados a ctr_k, então ctr_k envia os dados a n_j. Novamente, uma aresta entre dois elementos que executam em um mesmo recurso tem custo zero.

Para os nós com comunicação externa devemos considerar a comunicação entre controladores. Considere um nó n_i dentro dos controladores ctr_4, ctr_3, ctr_2, ctr_1, com $n_i \in ctr_1 \in ctr_2 \in ctr_3 \in ctr_4$. O custo de comunicação entre um nó $n_j \in ctr_4$ e n_i é $c_{j,ctr4} + c_{ctr4,ctr3} + c_{ctr3,ctr2} + c_{ctr2,ctr1} + c_{ctr1,i}$. Então, se cada um desses controladores estiver em um recurso diferente, o custo de comunicação pode ser proibitivo, tornando a execução seqüencial das tarefas mais eficiente. Ainda, quando um controlador ctr_k está sendo escalonado, o controlador que contém ctr_k ainda não foi escalonado, sendo desconhecida a largura de banda entre eles. Para a comunicação que entra ou sai de ctr_k, a comunicação considerada é aquela entre ctr_k e o nó externo, já escalonado, que comunica com ctr_k. O recurso selecionado para executar um controlador é aquele que minimiza a soma das comunicações dos nós nas duas classes. O escalonamento de controladores é mostrado no Algoritmo 8.

6.4.2 Exemplo de Execução

Na Figura 5.4, o primeiro controlador a ser escalonado é uma escolha aleatória entre os controladores 1, 2 e 3. Vamos assumir que o controlador 1 é escolhido. Se ele for atribuído ao recurso 0, o custo de comunicação seria nulo, já que todos os nós que comunicam com o controlador 1 (nós 5, 7, 8 e 10) estão escalonados no recurso 0. Se o controlador 0 for atribuído ao recurso 1, o custo de comunicação seria $c_{n_5,ctr_1} + c_{ctr_1,n_7} + c_{n_5,ctr_1} + c_{ctr_1,n_8} + c_{n_7,ctr_1} + c_{ctr_1,n_{10}} + c_{n_8,ctr_1} + c_{ctr_1,n_{10}} = 152.0$. Para os recursos 2 e 3, a comunicação seria de 304.5. Desta forma, o controlador 1 é escalonado no recurso 0. Seguindo o mesmo raciocínio, o controlador 2 é escalonado no recurso 0 e o controlador 3 no recurso 1. O controlador 4 é escalonado no recurso 0, pois todos os nós que comunicam com ele estão escalonados nesse recurso. O controlador 5 no recurso 0 teria comunicação de 55.0, no recurso 1 de 85.0 e de 280.0 nos recursos 2 ou 3. Então, o controlador 5 é atribuído ao recurso 0. Finalmente, o controlador 6 é escalonado no recurso 0, pois ele comunica somente com o controlador 5.

Algoritmo 8 escalona_controladores

```
1: while existem controladores não escalonados do
2:   ctr_k ⇐ controlador sem subcontroladores não escalonados
3:   for all r ∈ recursos do
4:     Atribui ctr_k a r
5:     for all n_k ∈ ctr_k do
6:       for all n_j que comunica com n_k do
7:         if n_j ∈ ctr_k then
8:           intComunic_r = intComunic_r + c_{n_k,n_j}
9:         else if n_j ∈ supercontrolador(ctr_k) then
10:          extComunic_r = extComunic_r + c_{n_j,ctr_k} + c_{ctr_k,n_k}
11:        end if
12:      end for
13:    end for
14:    nósComunic_r = intComunic_r + extComunic_r
15:    for all ctr_s ∈ subcontroladores(ctr_k) do
16:      subComunic_r = subComunic_r + c_{ctr_k,ctr_s} + c_{ctr_s,ctr_k}
17:    end for
18:    comunic_r ⇐ nósComunic_r + subComunic_r
19:  end for
20:  Escalona ctr_k em recurso( min_{r∈recursos} (comunic_r))
21: end while
```

6.5 O Algoritmo *Path Clustering Heuristic*

O conjunto dos passos apresentados – seleção de tarefas e agrupamento, seleção de recursos e escalonamento de controladores – forma o *Path Clustering Heuristic* (PCH), que é mostrado no Algoritmo 9. O primeiro passo do algoritmo é atribuir o DAG do processo ao sistema homogêneo virtual, para que seja possível calcular os atributos iniciais das tarefas. Em seguida o algoritmo inicia a iteração sobre os nós do grafo, criando os *clusters* e escalonando-os até que não restem nós do processo não escalonados. A cada *cluster* escalonado, o algoritmo recalcula os atributos *Weight*, EST e EFT dos nós. Após escalonar todos os nós e finalizar a iteração, o algoritmo realiza o escalonamento de controladores, concluindo o processo de escalonamento.

No escalonamento resultante para o grafo da Figura 5.4, apresentado na Figura 6.1, a tarefa 11 precisa esperar pelos dados computados pelas tarefas 9 e 3, sendo estas, pois, determinantes no tempo de início da tarefa 11. Os dados da tarefa 10 podem ser utilizados pela tarefa 11 assim que aquela termine, pois as tarefas 10 e 11 e seus controladores estão no mesmo recurso. O escalonamento retornado pelo PCH nesse caso gera *overhead* nulo de comunicação dos controladores.

Algoritmo 9 PCH

1: Atribui DAG ao sistema virtual homogêneo
2: Calcula atributos iniciais das tarefas
3: **while** existem nós não escalonados **do**
4:     *cluster* ⇐ gera_próximo_agrupamento()
5:     *resource* ⇐ seleciona_melhor_recurso(*cluster*)
6:     Escalona *cluster* em *recurso*
7:     Recalcula Weights, ESTs e EFTs
8: **end while**
9: escalona_controladores()

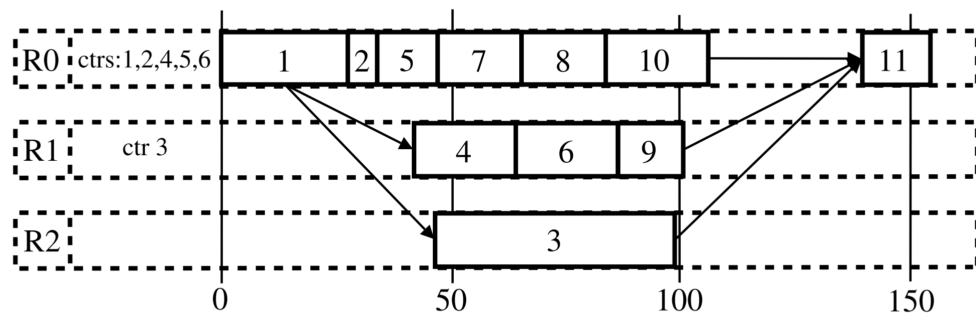

Figura 6.1: Escalonamento de tarefas e controladores para o grafo da Figura 5.4

.

6.6 Busca de Recursos em Grupos Adjacentes

Durante o escalonamento de tarefas o escalonador tem disponível um repositório com informações sobre todos os recursos do grupo e informações sobre parte dos recursos dos grupos adjacentes. Existem diferentes formas de determinar quais recursos serão disponibilizados aos grupos adjacentes. Por exemplo, podemos determinar que cada grupo envie a seus adjacentes informações sobre 10% de seus recursos, escolhendo os recursos com menor tempo de fila de execução. Também é possível determinar que cada grupo envie informações sobre 20% de seus recursos, escolhidos aleatoriamente. A forma escolhida para os experimentos realizados com o PCH determina que cada grupo envia aos grupos adjacentes informações sobre 20% dos recursos, escolhendo aqueles que têm maior capacidade de processamento dentro do grupo. Assim, cada grupo tem disponível outros recursos que poderiam ser utilizados pelo escalonador para diminuir o *makespan* dos escalonamentos realizados. Esse valor foi escolhido porque tendo informações sobre 20% dos melhores recursos de um grupo, o algoritmo tem um bom indício sobre a capacidade de processamento dos recursos de tal grupo. Dessa forma, é possível que o algoritmo conjecture, através da estratégia escolhida, se uma busca de mais recursos em um grupo

adjacente pode ser vantajoso. Uma análise experimental dessa porcentagem é mostrada na Seção 7.4.4.

Mesmo com algumas informações sobre recursos de grupos adjacentes, os recursos disponíveis no repositório podem não atender à demanda por computação dentro do grupo. Nesse caso um grupo pode solicitar a grupos adjacentes informações sobre outros recursos disponíveis. Para que a busca de recursos extras seja útil para a minimização dos *makespans* dos processos, primeiramente é necessário definir quando o algoritmo deve solicitar a um GM adjacente informações sobre outros recursos disponíveis. Com esse critério definido, o próximo passo é determinar em qual grupo adjacente, se existirem vários, a solicitação de informações deve ser feita.

O critério para busca deve determinar que sejam procurados novos recursos em grupos adjacentes quando os recursos do repositório não atenderem às necessidades de computação do processo, seja por falta de capacidade de processamento dos recursos ou por escassez de recursos no repositório. Se o critério não for definido de forma adequada, o escalonador pode executar pedidos de recursos que não têm efeito no escalonamento do processo. Além disso, solicitar recursos à revelia pode interferir no desempenho do sistema.

Novamente, é possível utilizar diferentes estratégias para determinar quando uma busca por recursos será realizada. Um exemplo é buscar recursos caso exista uma tarefa que demanda muita computação, na tentativa de encontrar um recurso com capacidade de processamento elevada que possa minimizar o tempo de execução da tarefa e do processo. Outro exemplo é realizar uma busca por recursos caso o tempo de fila dos recursos locais estejam acima de um limite pré-determinado. Para o PCH definimos que um grupo solicitará informações sobre recursos adicionais caso o recurso escolhido para um *cluster* já esteja alocado para outro *cluster* do mesmo processo. Essa pode ser uma indicação de que o grupo está com poucos recursos disponíveis ou a maioria dos recursos está com uma fila de execução grande, conseqüentemente diminuindo a capacidade de minimização do tempo de execução de um processo. Assim, se no passo de escolha de recurso para um *cluster* o recurso retornado já possui um *cluster* do mesmo processo, o PCH solicita informações sobre recursos adicionais de grupos adjacentes.

Complementar à definição de quando realizar-se-á uma busca, está a escolha do grupo para o qual a busca será direcionada. No PCH definimos que a solicitação é feita para o grupo adjacente que possuir o recurso com melhor capacidade de processamento no repositório do grupo solicitante. A partir do momento que os novos recursos são adicionados ao repositório, o escalonador está apto a utilizá-los no escalonamento. O *cluster* é re-escalonado, agora com novos recursos disponíveis no repositório. Se novamente o recurso escolhido já possuir um *cluster* do mesmo processo, o algoritmo repete a solicitação de novos recursos, desta vez para o segundo grupo que possuir o melhor recurso no repositório

do grupo solicitante. O *cluster* é re-escalonado, desta vez definitivamente, sem solicitar mais informações a outros grupos.

O Algoritmo 10 mostra as alterações necessárias no PCH para introduzir a busca por recursos em grupos adjacentes.

Algoritmo 10 PCH com busca de recursos

```
1:  Atribui DAG ao sistema virtual homogêneo
2:  Calcula atributos iniciais das tarefas
3:  while existem nós não escalonados do
4:     cluster ⇐ gera_próximo_agrupamento()
5:     resource ⇐ seleciona_melhor_recurso(cluster)
6:     if resource possui cluster do mesmo processo then
7:        grupo_adj ⇐ grupo adjacente com recurso de maior capacidade de processamento
8:        Solicita mais informações de recursos a grupo_adj e atualiza repositório
9:        resource ⇐ seleciona_melhor_recurso(cluster)
10:       if resource possui cluster do mesmo processo then
11:          grupo_adj2 ⇐ segundo grupo adjacente com recurso de maior capacidade de
             processamento
12:          Solicita mais informações de recursos a grupo_adj2 e atualiza repositório
13:          resource ⇐ seleciona_melhor_recurso(cluster)
14:       end if
15:    end if
16:    Escalona cluster em recurso
17:    Recalcula Weights, ESTs e EFTs
18: end while
19: escalona_controladores()
```

6.7 Análise de Complexidade

Um algoritmo de escalonamento, além de escalonar as tarefas minimizando o tempo de execução, deve também ter complexidade de tempo baixa. Sejam dois algoritmos A e B, onde A gera um escalonamento com *makespan* X e B gera escalonamento com *makespan* $X + Y$ para um processo P. Considere que B consome um tempo Z para escalonar o processo P. Se o tempo que A leva para escalonar o processo P for maior que $Y + Z$, então o algoritmo A não é mais eficiente que o algoritmo B, apesar de A gerar *makespans* menores. Não é aceitável que o tempo ganho nos *makespans* gerados pelo algoritmo de escalonamento seja perdido durante a execução deste.

Nesta seção analisamos a complexidade do algoritmo proposto para o conjunto dos grafos que podem ser gerados pelo modelo de programação do Xavantes. De fato, esta é a

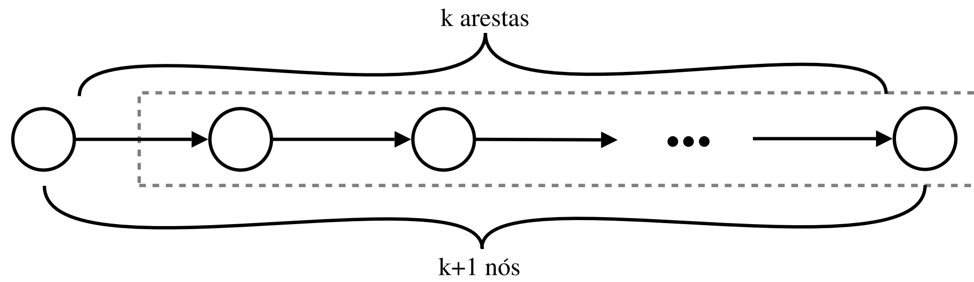

Figura 6.2: Grafo G': G após adição de um controlador seqüencial com k tarefas.

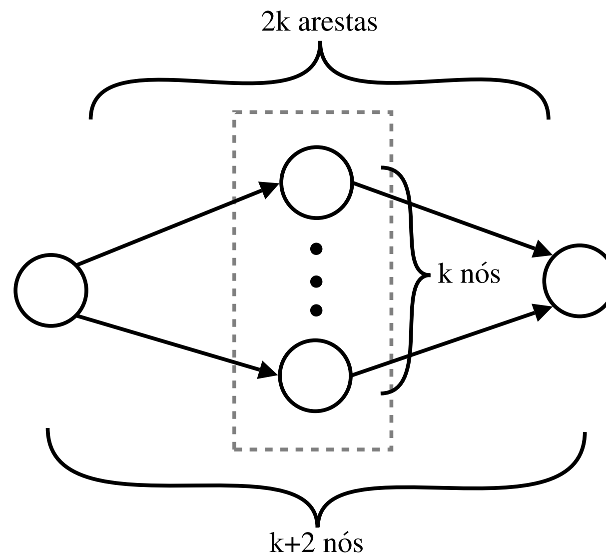

Figura 6.3: Grafo G': G após adição de um controlador paralelo com k tarefas.

análise que deve ser levada em consideração quando da execução do algoritmo proposto.

Seja um grafo acíclico direcionado $G = (N, E)$, com $|N| = 1$, conseqüentemente $|E| = 0$. O grafo G pode ser gerado pelo modelo de programação do Xavantes. Seja um grafo G' que é composto pelo grafo G e k novos nós. Para que G' seja um grafo concebível pelo modelo de programação, os próximos nós a serem adicionados devem ser especificados dentro de um controlador, de acordo com a forma estruturada do modelo. Se o próximo controlador especificado no processo for um controlador seqüencial com k tarefas, então G' passará a ter $|N| = k + 1$ e $|E| = k$, como mostra a Figura 6.2.

Se o próximo controlador especificado no processo for um controlador paralelo com k tarefas, então G passará a ter $|N| = k + 2$ e $|E| = 2k$, pois além das k tarefas do controlador haverá uma tarefa para a junção dos resultados das tarefas do controlador paralelo. A Figura 6.3 mostra esse passo.

O novo grafo G', composto a partir de um único nó com a inclusão das tarefas de um controlador paralelo ou seqüencial, será um grafo que pode ser gerado pelo modelo de programação. Seja um grafo G'' que é composto pelo grafo G' e m novos nós. Os m novos nós são adicionados no grafo através de um controlador especificado no modelo de programação, seqüencialmente ou internamente ao controlador que contém os k nós adicionados anteriormente ao grafo inicial G. Temos seis casos possíveis para a adição dos m novos nós:

1. G' composto por um controlador *seqüencial* e os m novos nós são adicionados ao processo através de um controlador *seqüencial* (Figura 6.4).

2. G' composto por um controlador *paralelo* e os m novos nós são adicionados ao processo através de um controlador *seqüencial* interno ao controlador paralelo (Figura 6.5).

3. G' composto por um controlador *paralelo* e os m novos nós são adicionados ao processo através de um controlador *seqüencial* externo o controlador paralelo (Figura 6.6).

4. G' composto por um controlador *seqüencial* e os m novos nós são adicionados ao processo através de um controlador *paralelo* (Figura 6.7).

5. G' composto por um controlador *paralelo* e os m novos nós são adicionados ao processo através de um controlador *paralelo*, interno ao controlador paralelo (Figura 6.8).

6. G' composto por um controlador *paralelo* e os m novos nós são adicionados ao processo através de um controlador *paralelo*, externo ao controlador paralelo (Figura 6.9).

A Figura 6.4 mostra o caso em que os nós adicionados fazem parte de um controlador seqüencial e são adicionados ao final do controlador seqüencial de G', o que resultaria em um processo G'' com $|N| = k + m + 1$ e $|E| = k + m$.

Se os m nós compõem um controlador seqüencial e são adicionados dentro de um controlador paralelo, então o grafo G'' teria $|N| = k(1 + m) + 2$ e $|E| = k(2 + m)$, como mostra a Figura 6.5.

A Figura 6.6 mostra o caso em que um controlador seqüencial é adicionado ao final de um controlador paralelo. Nesse caso, G'' teria $|N| = k + m + 2$ e $|E| = 2k + m$.

Se os m nós adicionados pertencerem a um controlador paralelo e forem adicionados ao processo após um controlador seqüencial, então G'' teria $|N| = k + m + 2$ e $|E| = k + 2m$, como mostra a Figura 6.7.

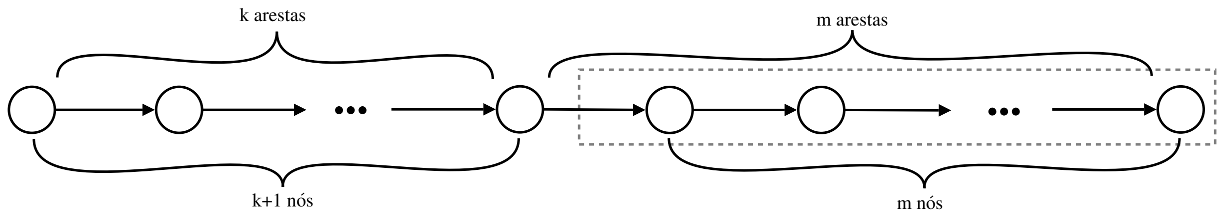

Figura 6.4: Grafo G'': G' após adição de um controlador seqüencial com m tarefas.

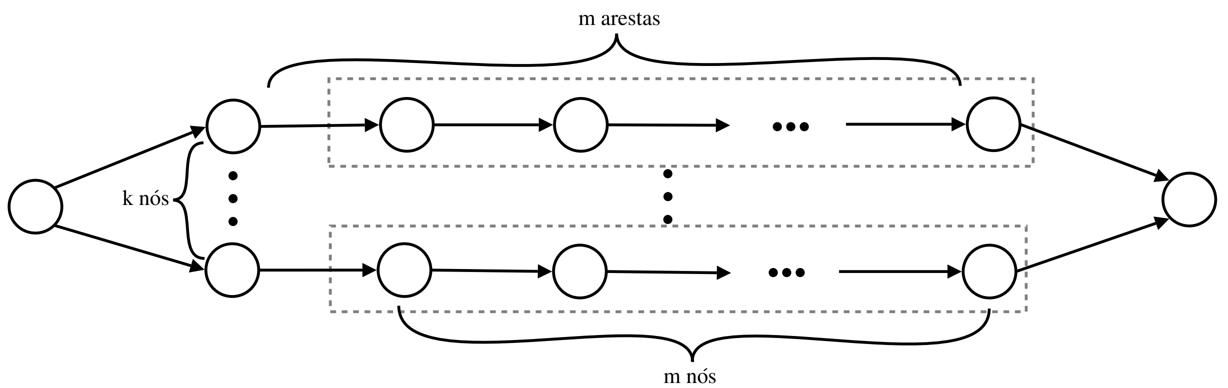

Figura 6.5: Grafo G'': G' após adição de um controlador seqüencial com m tarefas.

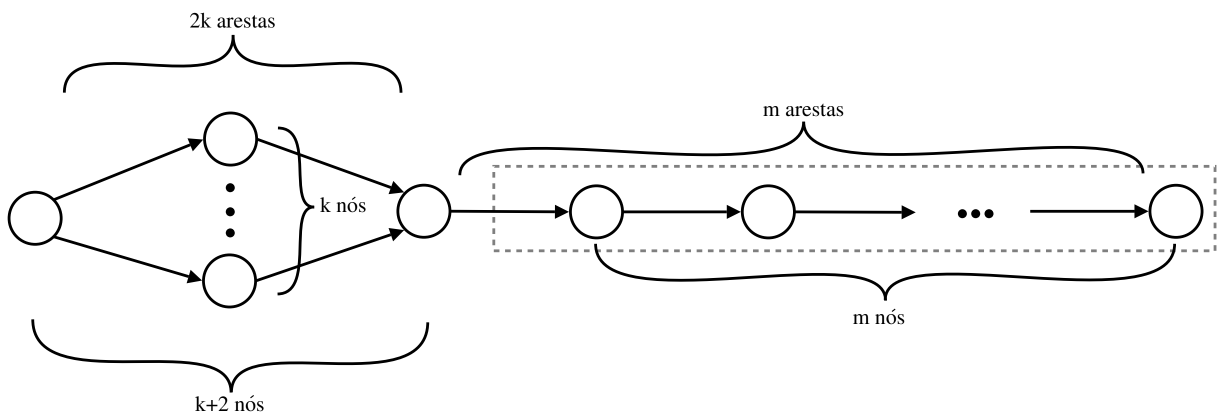

Figura 6.6: Grafo G'': G' após adição de um controlador seqüencial com m tarefas.

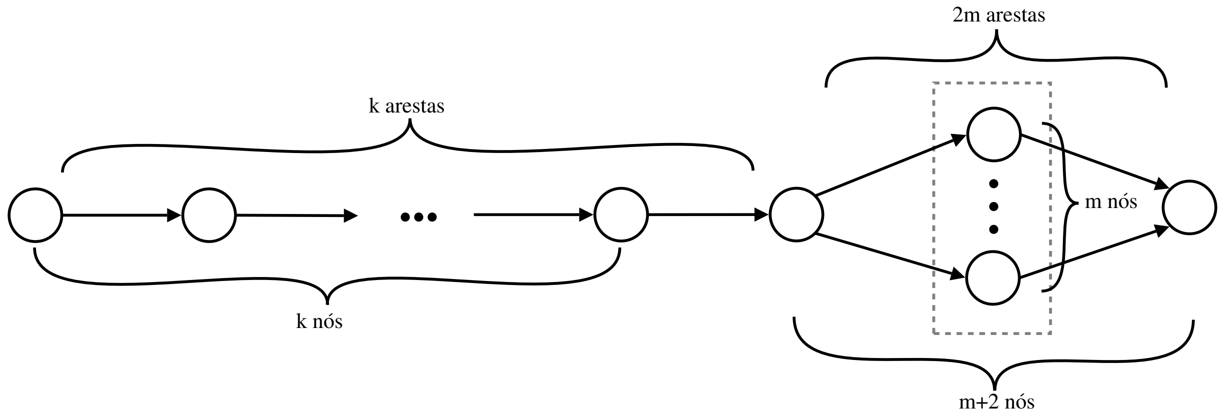

Figura 6.7: Grafo G'': G' após adição de um controlador paralelo com m tarefas.

A Figura 6.8 mostra o caso em que o controlador adicionado contém m nós em paralelo, e esse controlador é inserido internamente a outro controlador paralelo. Assim, G'' teria $|N| = k(m+2)+2$ e $|E| = 2(k+km)$.

O caso em que o controlador adicionado contém m nós em paralelo é inserido seqüencialmente a outro controlador paralelo é mostrado na Figura 6.9. Nesse caso, G'' teria $|N| = k+m+3$ e $|E| = 2(k+m)$.

Se continuarmos adicionando controladores ao processo, os passos anteriores se repetirão e a adição de cada novo controlador poderá ser representada por um dos casos citados. Note que para todos os casos $|N| \leq 2|E|$:

- $|N| = k+1 <= 2|E| = 2k$
- $|N| = k+2 < 2|E| = 2(2k) = 4k$
- $|N| = k+m+1 < 2|E| = 2(k+m) = 2k+2m$
- $|N| = k(1+m)+2 < 2|E| = 2(k(2+m)) = 2k(2+m)$
- $|N| = k+m+2 < 2|E| = 2(2k+m) = 4k+2m$
- $|N| = k+m+2 < 2|E| = 2(k+2m) = 2k+4m$
- $|N| = k(m+2)+2 < 2|E| = (2(2(k+km)) = 4(k+km)$
- $|N| = k+m+3 < 2|E| = (2(2(k+m)) = 4k+4m$

Durante o desenvolvimento de um processo utilizando o modelo de programação do Xavantes, cada controlador adicionado terá um correspondente em um dos casos citados na análise realizada. Assim, um processo com n tarefas representado pelo modelo de

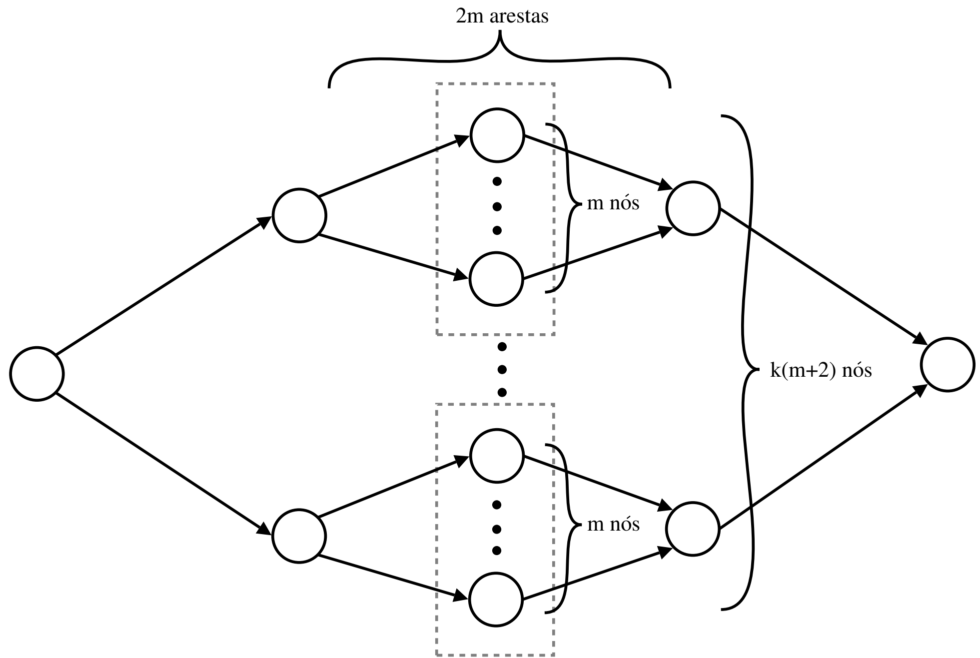

Figura 6.8: Grafo G'': G' após adição de um controlador paralelo com m tarefas.

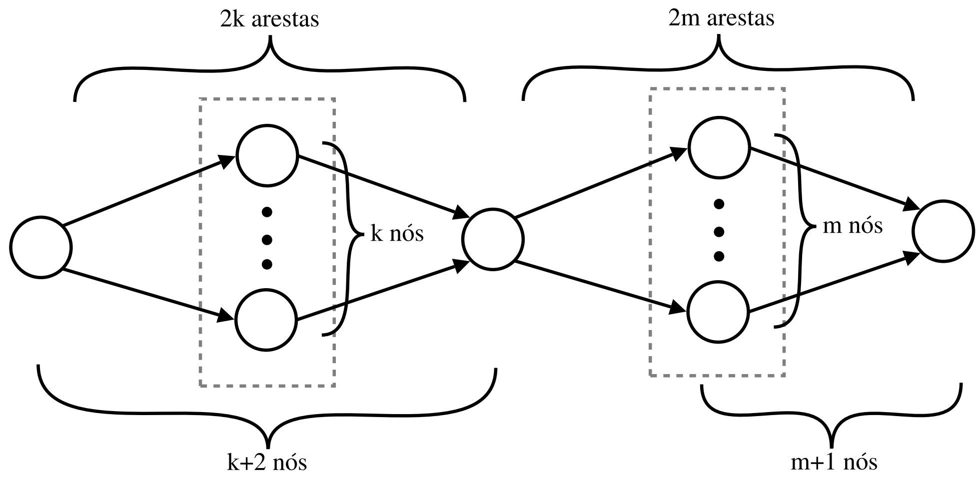

Figura 6.9: Grafo G'': G' após adição de um controlador paralelo com m tarefas.

programação do Xavantes terá no máximo $2n$ arestas. Em outras palavras, o número de arestas de um processo desenvolvido sobre o modelo de programação Xavantes é da ordem de n, onde n é o número de nós do grafo.

Para analisarmos a complexidade de tempo do algoritmo como um todo, devemos analisar cada linha de cada passo do algoritmo. Na análise aqui apresentada, ao lado de cada linha está o tempo necessário para executá-la. O tempo mostrado ao lado do início de cada iteração é referente ao número de vezes que tal iteração pode ocorrer. Dessa forma, o tempo total de computação de um iteração é igual ao maior tempo interno à iteração multiplicado pelo número de vezes que a iteração pode ocorrer.

No Algoritmo 6, passo do PCH que gera agrupamentos de caminhos, temos:

1: $n \Leftarrow$ nó não escalonado com a maior Prioridade {**Tempo** $O(1)$}
2: $cluster \Leftarrow cluster \cup n$ {**Tempo** $O(1)$}
3: **while** (n tem sucessores não escalonados) **do** {**Tempo** $O(n)$}
4: $n_{suc} \Leftarrow sucessor_i$ de n com maior $P_i + EST_i$ {**Tempo** $O(1)$}
5: $cluster \Leftarrow cluster \cup n_{suc}$ {**Tempo** $O(1)$}
6: $n \Leftarrow n_{suc}$ {**Tempo** $O(1)$}
7: **end while**
8: **return** $cluster$ {**Tempo** $O(1)$}

Ao receber um processo, o algoritmo de escalonamento pode criar uma lista de nós não escalonados ordenada de forma não crescente por prioridade. A criação da lista leva tempo $O(n \log n)$ e é feita apenas uma vez durante o escalonamento de um processo, antes de iniciar a iteração sobre os nós. Então, na linha 1 do algoritmo que gera agrupamentos é possível selecionar o nó não escalonado com a maior prioridade em tempo constante $O(1)$. A linha 2 é executada trivialmente, sendo a adição do nó n ao conjunto vazio $cluster$. A iteração da linha 3, no pior caso, ocorre n vezes. Esse caso acontece quando o grafo não tem elementos paralelos, apenas tarefas seqüenciais. A linha 4, analogamente à linha 1, pode ser executada em tempo constante $O(1)$, utilizando uma fila ordenada. A linha 5 também é executada em tempo constante, como a linha 2. Também são computadas em tempo constante as linhas 6 e 8. Então, o Algoritmo 6 leva tempo $O(n)$, no pior caso, para criar um agrupamento de acordo com a estratégia do algoritmo proposto.

O próximo passo do algoritmo é a seleção de recursos, mostrada no Algoritmo 7. A complexidade de tempo de cada linha desse passo é a seguinte:

1: **for all** $r \in recursos$ **do** {**Tempo** $O(r)$}
2: $escalonamento \Leftarrow$ Insere $cluster$ em $escalonamento_r$ {**Tempo** $O(n)$}
3: calcula_EFT($escalonamento$) {**Tempo** $O(n)$}
4: $tempo_r \Leftarrow$ calcula_EST(sucessor($cluster$)) {**Tempo** $O(1)$}

5: **end for**
6: **return** recurso r com menor $tempo_r$ {**Tempo** $O(1)$}

Na segunda linha do algoritmo de seleção de recursos é realizada uma união entre duas listas ordenadas que resulta em uma terceira lista ordenada. Para realizar essa operação são necessários $O(n)$ passos. Para calcular o EFT de cada tarefa escalonada em um recurso, na linha 3 do algoritmo, o tempo é de $O(n)$. Na linha 4 é feito o cálculo do EST do sucessor do *cluster* que está sendo escalonado em tempo constante $O(1)$, o mesmo tempo da linha número 6. A iteração desse trecho de código é realizada r vezes, então a complexidade de tempo do passo de seleção de recursos é $O(nr)$ no pior caso.

O algoritmo de escalonamento de controladores é determinante no *makespan* final do escalonamento das tarefas, pois objetiva suprimir a comunicação entre tarefas e controladores. Esse é um passo executado após o término do escalonamento de tarefas. A complexidade de tempo do código do escalonador de controladores é como segue:

1: **while** existem controladores não escalonados **do** {**Tempo** $O(n)$}
2: $ctr_k \Leftarrow$ controlador sem subcontroladores não escalonados {**Tempo** $O(1)$}
3: **for all** $r \in recursos$ **do** {**Tempo** $O(r)$}
4: Atribui ctr_k a r {**Tempo** $O(1)$}
5: **for all** $n_k \in ctr_k$ **do** {**Tempo** $O(n)$}
6: **for all** n_j que comunica com n_k **do** {**Tempo** $O(n)$}
7: **if** $n_j \in ctr_k$ **then** {**Tempo** $O(1)$}
8: $intComunic_r = intComunic_r + c_{n_k,n_j}$ {**Tempo** $O(1)$}
9: **else if** $n_j \in$ supercontrolador(ctr_k) **then** {**Tempo** $O(1)$}
10: $extComunic_r = extComunic_r + c_{n_j,ctr_k} + c_{ctr_k,n_k}$ {**Tempo** $O(1)$}
11: **end if**
12: **end for**
13: **end for**
14: $nósComunic_r = intComunic_r + extComunic_r$ {**Tempo** $O(1)$}
15: **for all** $ctr_s \in$ subcontroladores(ctr_k) **do** {**Tempo** $O(n)$}
16: $subComunic_r = subComunic_r + c_{ctr_k,ctr_s} + c_{ctr_s,ctr_k}$ {**Tempo** $O(1)$}
17: **end for**
18: $comunic_r \Leftarrow nósComunic_r + subComunic_r$ {**Tempo** $O(1)$}
19: **end for**
20: Escalona ctr_k em recurso($\min\limits_{r \in recursos}(comunic_r)$) {**Tempo** $O(1)$}
21: **end while**

No código referente ao escalonamento de controladores as linhas 2, 4, 8, 10, 14, 16, 18 e 20 efetuam computações diretas, portanto consomem tempo constante $O(1)$. As

linhas que especificam as iterações são as que determinam a complexidade desse trecho de código. A iteração da linha 1 é executada $O(n)$ vezes. Na linha 3 há uma iteração sobre os recursos disponíveis, então com tempo $O(r)$. O tempo consumido pela iteração da linha 5 é $O(n)$. A iteração da linha 6 é dependente do número de arestas do nó sobre o qual a iteração está executando. Como mostrado anteriormente, o número máximo de arestas em um grafo gerado pelo modelo de programação do Xavantes é da ordem de n, onde n é o número de nós do grafo. Assim, essa iteração consome tempo $O(n)$, no pior caso. Então, no algoritmo que escalona os controladores temos 4 iterações aninhadas, 3 com tempo $O(n)$ e 1 com tempo $O(r)$, além da iteração da linha 15, com tempo $O(n)$. A complexidade do escalonamento de controladores é $O(rn^3)$.

Para finalizar a análise de complexidade, temos o algoritmo final de escalonamento:

1: Atribui DAG ao sistema virtual homogêneo {**Tempo** $O(n)$}
2: Calcula atributos iniciais das tarefas {**Tempo** $O(n)$}
3: **while** existem nós não escalonados **do** {**Tempo** $O(n)$}
4:     *cluster* ⇐ gera_próximo_agrupamento() {**Tempo** $O(n)$}
5:     *resource* ⇐ seleciona_melhor_recurso(*cluster*) {**Tempo** $O(rn)$}
6:     Escalona *cluster* em *recurso* {**Tempo** $O(n)$}
7:     Recalcula Weights, ESTs e EFTs {**Tempo** $O(n)$}
8: **end while**
9: escalona_controladores() {**Tempo** $O(rn^3)$}

Na linha 1, o tempo necessário para a atribuição inicial das tarefas ao sistema virtual homogêneo é $O(n)$, assim como o cálculo inicial dos atributos das tarefas do grafo. A iteração da linha 3 pode ocorrer até n vezes. A linha 4 consome tempo $O(n)$, como explanado anteriormente. O tempo de computação na linha 5 é $O(rn)$. O escalonamento do agrupamento de tarefas feito na linha 6 leva tempo $O(n)$, da mesma forma que a computação dos atributos na linha 7. Dessa forma, a iteração da linha 3 consome tempo $O(rn^2)$. Externo à iteração, o escalonamento de controles na linha 9 determina a complexidade de tempo do algoritmo como um todo. A importância do escalonamento de controladores na execução de processos no Xavantes é evidenciada, consumindo tempo $O(rn^3)$.

A complexidade de tempo do PCH é $O(rn^3)$, sendo esta aceitável e condizente com o problema de escalonamento de tarefas. A complexidade apresentada pelo PCH é semelhante à complexidade apresentada por outros algoritmos de escalonamento de tarefas em sistemas heterogêneos.

Capítulo 7

Resultados Experimentais

O algoritmo aqui proposto foi desenvolvido com o intuito de ter bom desempenho no escalonamento do middleware Xavantes. A comparação mostrada neste capítulo justifica o desenvolvimento de um novo algoritmo em detrimento da utilização de algoritmos de escalonamento existentes, além de mostrar que o algoritmo proposto, o PCH, tem bom desempenho e é escalável na arquitetura do middleware Xavantes. Comparamos o algoritmo PCH com o algoritmo HEFT [26] e com o algoritmo CPOP [26]. Quinze grafos válidos no modelo de programação do Xavantes foram gerados aleatoriamente para o experimento. Cada grafo foi escalonado 1000 vezes utilizando cada algoritmo, com valores aleatórios de uma distribuição uniforme em nós e arestas. Os recursos considerados são heterogêneos, assim como a largura de banda entre eles. Os três algoritmos foram executados com e sem controladores, variando os custos de comunicação e computação. Comunicação baixa significa que todos os custos de comunicação são menores que todos os custos de computação. Comunicação média significa que os custos de computação e comunicação são gerados aleatoriamente dentro do mesmo intervalo. Comunicação alta significa que todos os custos de comunicação são maiores que todos os custos de computação. O algoritmo utilizado para o escalonamento de controladores nos cenários dos algoritmos HEFT, CPOP e Proposto é o mesmo.

Para a comparação dos resultados entre o PCH, o algoritmo HEFT e o algoritmo CPOP, as métricas *SLR*, *Speedup* e *número de melhores escalonamentos* foram utilizadas.

7.1 Métricas de Desempenho

A análise de algoritmos de escalonamento de tarefas é feita através de simulações, já que trata-se de um problema sem solução conhecida em tempo polinomial e o desempenho do escalonador depende de variáveis alheias ao seu escopo. As principais métricas de desempenho para algoritmos de escalonamento utilizadas na literatura são o *Schedule*

Length Ratio (SLR) e o *Speedup*. O SLR é dado por:

$$SLR = \frac{makespan}{\displaystyle\sum_{n_i \in CC} \frac{instruções_{n_i}}{processamento_{melhor}}}$$

onde, CC é o conjunto de nós do caminho crítico do grafo inicial e $processamento_{melhor}$ é a capacidade de processamento do melhor recurso disponível. Assim, a soma no denominador representa o custo de computação do caminho crítico do grafo no melhor recurso disponível. O *Speedup* é dado por:

$$Speedup = \frac{\displaystyle\sum_{n_i \in V} \frac{instruções_{n_i}}{processamento_{melhor}}}{makespan}$$

O somatório no numerador representa o tempo de execução seqüencial de todas as tarefas no melhor recurso disponível. O número de vezes que um algoritmo resulta em escalonamentos com menor *makespan* também é uma métrica bastante utilizada. Essa métrica deve ser utilizada complementarmente a outras, já que apenas indica se um escalonamento é melhor que outro, mas não esclarece o quanto.

7.2 O Algoritmo HEFT

O algoritmo Heterogeneous Earliest Finish Time (HEFT) [26] utiliza a técnica de *list scheduling* para escalonar tarefas dependentes em sistemas heterogêneos. Os custos de computação e comunicação são considerados diferentes em cada recurso, então o HEFT inicializa os custos de computação e comunicação dos nós e arestas com os valores médios dos custos em todos os recursos. Então, HEFT computa o valor $rank_u$ para cada tarefa, percorrendo o grafo de trás para frente, com início na última tarefa do grafo (nó de saída). A tarefa pronta não escalonada com o maior valor $rank_u$ é selecionada e escalonada no processador que proporciona o menor tempo estimado de término. Durante o processo de escolha do processador, o algoritmo HEFT procura por janelas de tempo entre as tarefas já escalonadas. Caso exista uma janela que minimize o tempo estimado de término e obedeça as precedências das tarefas, então o processador que possui tal janela é selecionado. A complexidade de tempo do HEFT é $O(rn^3)$ [16]. O Algoritmo 11 mostra o pseudo-código do HEFT [26].

Algoritmo 11 HEFT

1: Inicializa os custos de comunicação e computação com valores médios
2: Computa $rank_u$ para todas as tarefas, percorrendo o grafo do fim para o início, partindo do nó de saída
3: Cria lista ordenada de tarefas em ordem não crescente de $rank_u$
4: **while** existem tarefas não escalonadas na lista **do**
5: Seleciona a primeira tarefa n_i da lista para escalonamento
6: **for all** processador p_k no conjunto de processadores **do**
7: Calcule $EFT(n_i, p_k)$ usando a política de escalonamento com busca de janelas
8: **end for**
9: Atribua a tarefa n_i ao processador p_j que minimiza EFT_{n_i}
10: **end while**

7.3 O Algoritmo CPOP

O algoritmo *Critical Path on a Processor* (CPOP) [26] também é um algoritmo de escalonamento de tarefas em sistemas heterogêneos. O CPOP também inicializa os custos de computação e comunicação com valores médios, além de utilizar o valor $rank_u$. Além do $rank_u$, o CPOP calcula o $rank_d$ para cada nó, percorrendo o grafo do nó de entrada até o nó de saída. Calculados esses dois valores, a prioridade de cada nó é calculada como $P_i = rank_u + rank_d$. Antes de iterar sobre os nós do grafo para promover o escalonamento, o algoritmo cria um conjunto que contém os nós que pertencem ao caminho crítico do grafo inicial. Então, o processador que minimiza o tempo de execução das tarefas do conjunto é selecionado para executá-las. Assim, o processo de escalonamento inicia selecionando primeiramente o nó de entrada. Como esse nó pertence ao caminho crítico do grafo inicial, ele é escalonado no processador escolhido para executar os nós do caminho crítico. O próximo nó selecionado será o nó de maior prioridade. Se esse nó não pertence ao caminho crítico, ele será escalonado no processador que minimizar seu tempo estimado de término. Caso contrário, ele é automaticamente escalonado no mesmo processador do nó de entrada. Esse processo se repete até que não existam mais nós não escalonados. O CPOP, que tem complexidade de tempo $O(rn^2)$, é mostrado no Algoritmo 12.

7.4 Resultados

As avaliações apresentadas inicialmente nesta seção são resultados de simulações realizadas considerando uma topologia de sistema heterogêneo. Essa topologia pode ser interpretada como um grupo isolado de recursos heterogêneos na arquitetura Xavantes, para validar a capacidade do algoritmo de escalonar tarefas em sistemas com recursos com desempenhos diferentes. Após os resultados em um grupo heterogêneo, apresentamos resultados em

Algoritmo 12 CPOP

1: Inicializa os custos de comunicação e computação com valores médios
2: Computa $rank_u$ para todas as tarefas, percorrendo o grafo de trás para frente, partindo do nó de saída
3: Computa $rank_d$ para todas as tarefas, percorrendo o grafo do início para o fim, partindo do nó de entrada
4: Computa $prioridade(n_i) = rank_u(n_i) + rank_d(n_i)$ para cada tarefa do grafo
5: $|CP| = prioridade(n_{entrada})$
6: $Conjunto_{CP} = \{n_{entrada}\}$, onde $Conjunto_{CP}$ é o conjunto de tarefas no caminho crítico
7: $n_k \Leftarrow n_{entrada}$
8: **while** n_k não é o nó de saída **do**
9: Seleciona n_j, onde $((n_j \in succ(n_k))$ and $(prioridade(n_j) == |CP|))$
10: $Conjunto_{CP} = Conjunto_{CP} \cup \{n_j\}$
11: $n_k \Leftarrow n_j$
12: **end while**
13: Seleciona o processador caminho crítico (p_{CP}) que minimiza $\sum_{n_i \in Conjunto_{CP}} w_{i,j}, \ \forall p_j$
14: Inicializa fila de prioridade com $n_{entrada}$
15: **while** existe uma tarefa não escalonada na fila de prioridade **do**
16: Seleciona a tarefa com maior prioridade, n_i, da fila de prioridade
17: **if** $n_i \in Conjunto_{CP}$ **then**
18: Atribui tarefa n_i ao processador (p_{CP})
19: **else**
20: Atribui a tarefa n_i ao processador p_j que minimiza $EFT(n_i, p_j)$.
21: **end if**
22: Atualiza fila de prioridade com os sucessores de n_i, caso estes se tornem *tarefas prontas*.
23: **end while**

sistemas com topologia de grupos, com o número de grupos e recursos variando dentro de uma faixa determinada. Para finalizar os experimentos apresentamos resultados com o PCH realizando busca de recursos em grupos adjacentes.

7.4.1 Sistema Heterogêneo

Para validar a capacidade de escalonamento do algoritmo em um sistema heterogêneo, os experimentos iniciais foram realizados considerado um sistema heterogêneo composto por recursos com poder de computação variável e *links* heterogêneos, com o número de recursos variando entre 2 e 7 e o número de nós dos grafos variando de 7 a 82. A topologia dos recursos nesse experimento é a de um grupo composto por máquinas heterogêneas ligadas

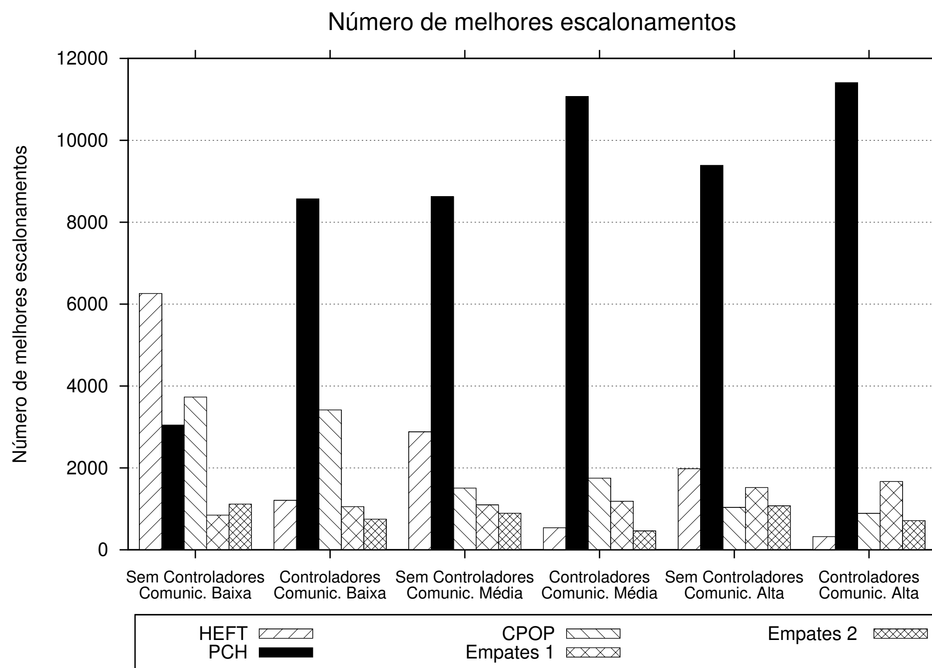

Figura 7.1: Número de escalonamentos com menor *makespan*.

por *links* heterogêneos, caracterizando apenas um grupo de recursos. Neste experimento não foi considerada a busca de recursos em grupos adjacentes pelo PCH.

A Figura 7.1, que apresenta o número de melhores escalonamentos gerados para cada algoritmo, mostra que o PCH gera escalonamentos melhores na maioria das execuções quando controladores são considerados, como esperado e desejado. Com comunicação baixa, em 57.15% das execuções o *makespan* do PCH foi menor. Com comunicações média e alta, essas taxas são de 73.81% e 76.06%, respectivamente. Quando controladores não são considerados, o algoritmo HEFT gera escalonamentos melhores quando a comunicação é baixa em 41.73% do total de execuções, contra 24.87% do CPOP e 20.32% do PCH. Com comunicação média e sem controladores, o PCH é melhor em 57.49% dos escalonamentos, enquanto HEFT é melhor em 19.19% e CPOP em 10.05% das execuções. Quando a comunicação é alta, o PCH é melhor em 62.6% dos casos, contra 13.21% do HEFT e 6.89% do CPOP. Os resultados de empate são representados de duas maneiras: *Empate 1* são escalonamentos onde o PCH gerou o melhor *makespan*, porém outro algoritmo também o fez, enquanto *Empate 2* são resultados onde CPOP e HEFT geraram escalonamentos iguais e melhores que o PCH.

A Figura 7.2 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 3 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é

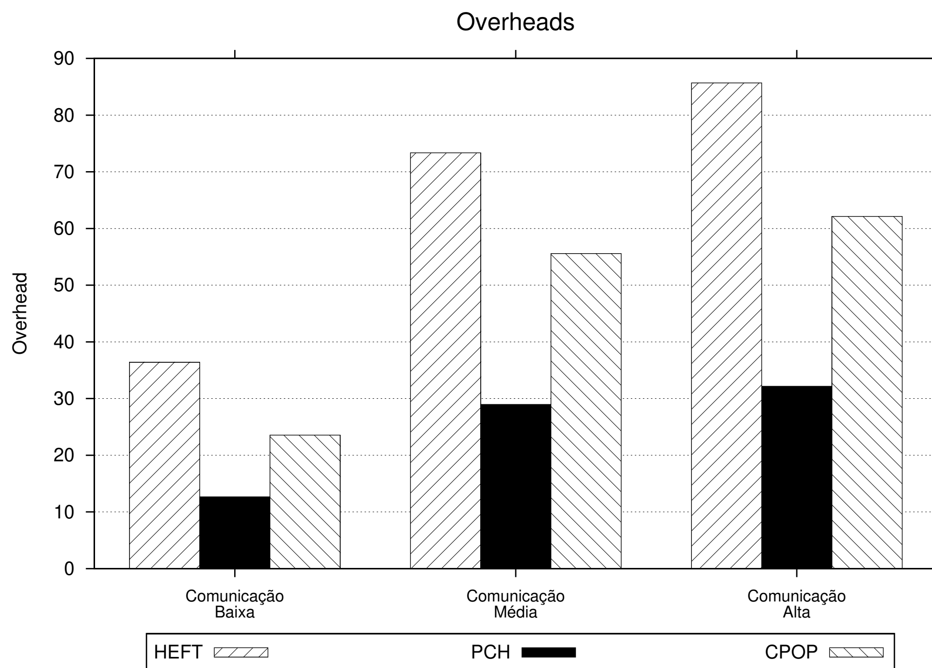

Figura 7.2: *Overhead* de comunicação dos controladores.

da ordem de 2 vezes o *overhead* do PCH, confirmando a afinidade do algoritmo proposto com a arquitetura Xavantes.

A Figura 7.3 mostra o SLR médio das execuções de cada algoritmo. Nessa figura podemos ver que a diferença entre os SLRs médios dos três algoritmos sem considerar controladores é baixa na maioria das execuções, com o PCH gerando SLRs menores nos cenários de comunicação média e alta. Quando consideramos controladores, essa diferença é maior, com o PCH resultando em SLRs menores em todos os casos. Isso significa que a maioria dos *makespans* gerados pelo algoritmo proposto é consideravelmente menor que os *makespans* gerados pelo HEFT e pelo CPOP.

O experimento desta seção mostra que, para recursos com topologia semelhante a de um grupo do Xavantes, o algoritmo PCH apresenta desempenho superior ao HEFT e ao CPOP sempre que controladores são considerados.

7.4.2 Topologia de Grupos

Nesta seção apresentamos resultados de simulações realizadas em um sistema com topologia de grupos heterogêneos, seguindo a arquitetura do middleware Xavantes, e sem busca por recursos em grupos adjacentes. Os cenários avaliados têm variação no número de grupos e número de recursos possíveis em cada grupo. Nos experimentos desta seção a

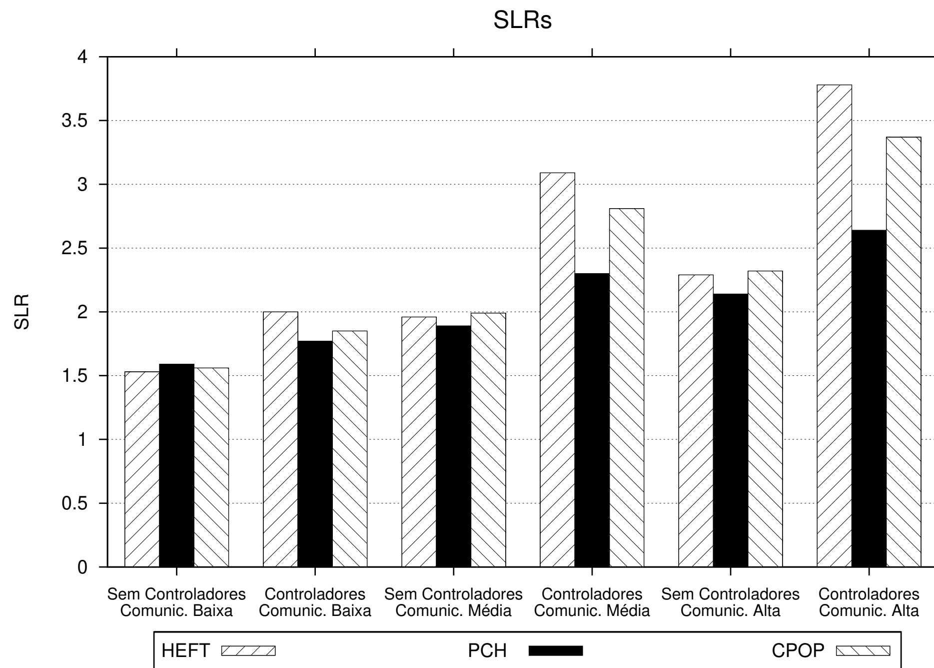

Figura 7.3: SLR médio.

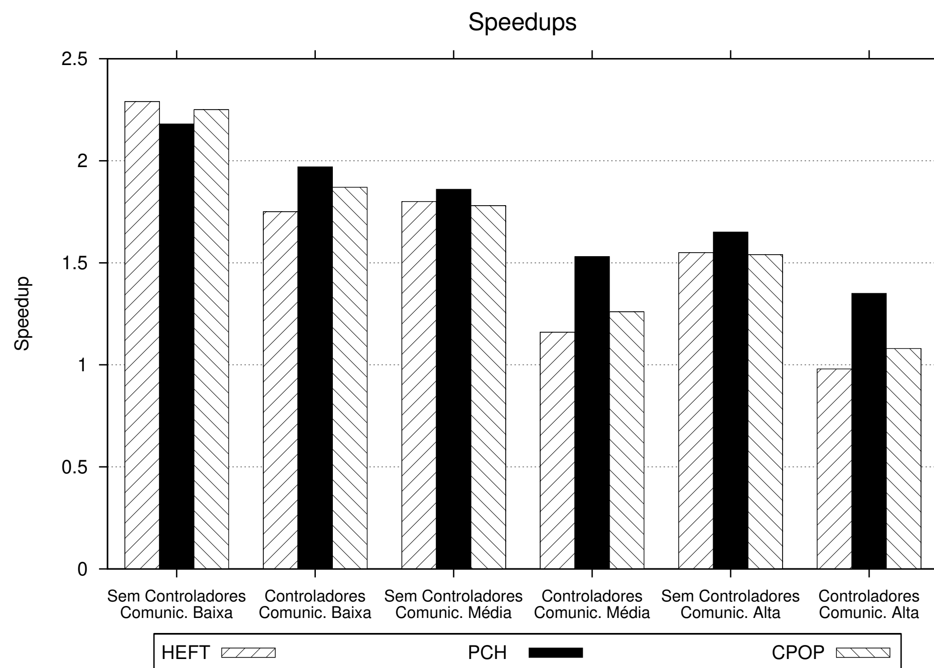

Figura 7.4: *Speedup* médio.

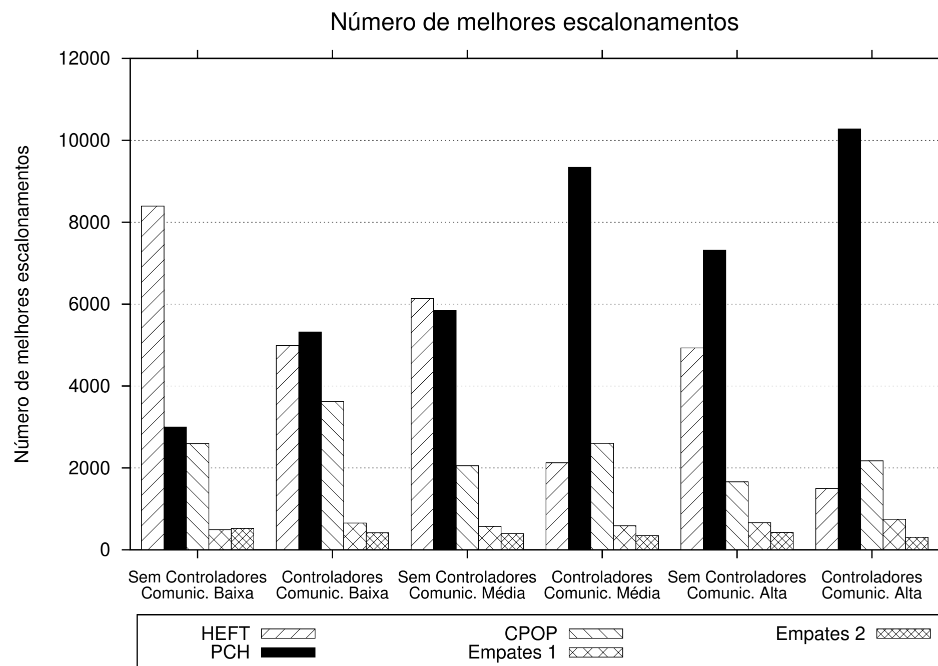

Figura 7.5: Número de escalonamentos com menor *makespan* em no máximo 2 grupos com no máximo 3 recursos.

variação aleatória é nos números de grupos e recursos, seguindo com as variações na capacidade de processamento, transmissão de dados, custos de computação e comunicação. A variação no número de grupos e de recursos permite avaliar a escalabilidade do PCH na arquitetura Xavantes, importante por se tratar de um algoritmo para grades computacionais. Para todo cenário apresentado nesta seção cada grafo foi escalonado 1000 vezes utilizando cada algoritmo, da mesma forma que no experimento da seção anterior.

A Figura 7.5 apresenta o número de melhores escalonamentos gerados para cada algoritmo em um cenário onde o número de grupos é no máximo 2 e cada grupo tem no máximo 3 recursos, variando aleatoriamente dentro desses limites a cada execução. Em comparação aos resultados da seção anterior, no cenário com limite de 2 grupos e 3 recursos há uma queda de desempenho em relação ao HEFT e ao CPOP. Nesse cenário e com comunicação baixa, o PCH resulta em melhores escalonamentos, absolutos ou em conjunto com outro algoritmo, em 39.83% das execuções quando controladores são considerados. Com controladores e comunicação média ou alta o PCH continua gerando melhores escalonamentos na maioria dos casos, 66.19% e 73.5% respectivamente.

A Figura 7.6 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 2 grupos com no máximo 3 recursos cada. Com comunicação baixa, o *overhead* nos escalonamentos gerados pelo HEFT é da

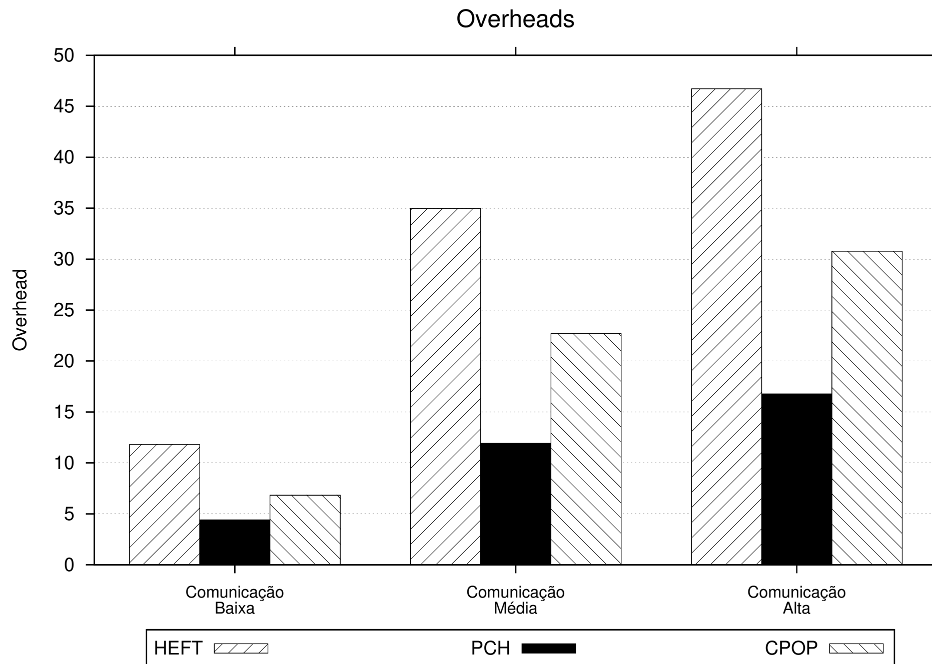

Figura 7.6: *Overhead* de comunicação dos controladores em no máximo 2 grupos com no máximo 3 recursos.

ordem de 2.67 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.54 vezes o *overhead* do PCH. Com comunicação média, HEFT tem *overhead* da ordem de 2.93 vezes o *overhead* do PCH, e CPOP tem *overhead* da ordem de 1.90 vezes àquele gerado pelo PCH. Com comunicação essa relação é de 2.79 para o HEFT e 1.83 para o CPOP.

A Figura 7.7 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 2 e número de recursos limitado a 3 por grupo. Sempre que controladores são considerados o PCH se mantém com a média de SLR melhor que HEFT e CPOP.

A Figura 7.8 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 2 grupos com no máximo 3 recursos cada. O *speedup* do PCH é inferior quando controladores não são considerados e a comunicação é baixa, confirmando o desempenho do PCH abaixo do HEFT neste cenário. Quando a comunicação é baixa, média ou alta com controladores, o PCH torna-se melhor que HEFT e CPOP para o limite de grupos e recursos considerado.

Considerando um cenário onde o número máximo de grupos é 2 e o número máximo de recursos por grupos é 10, a Figura 7.9 apresenta o número de vezes que cada algoritmo gerou o melhor escalonamento. Em relação ao cenário anterior, com no máximo 2 grupos

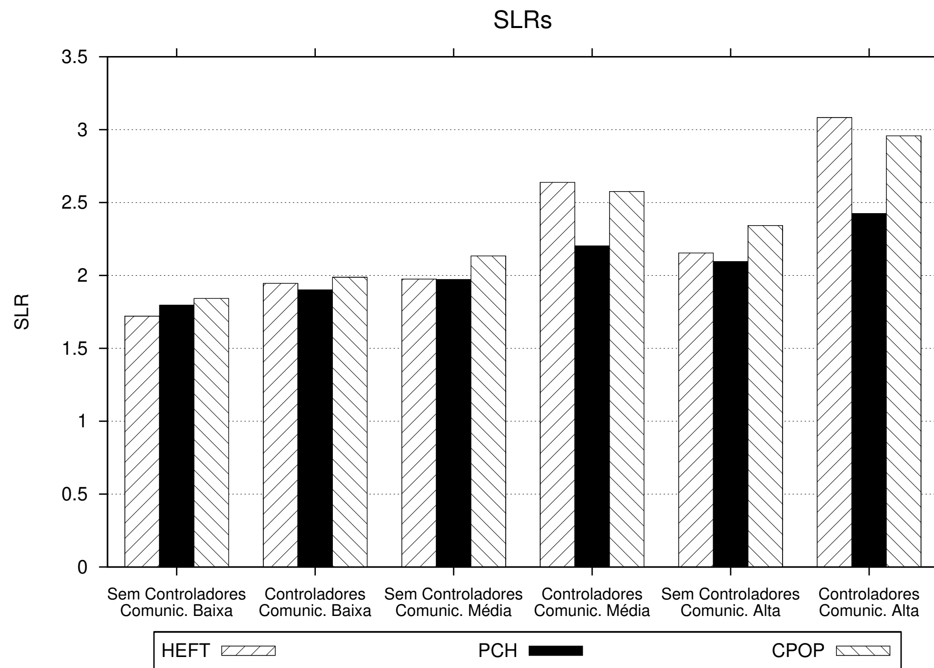

Figura 7.7: SLR médio em no máximo 2 grupos com no máximo 3 recursos.

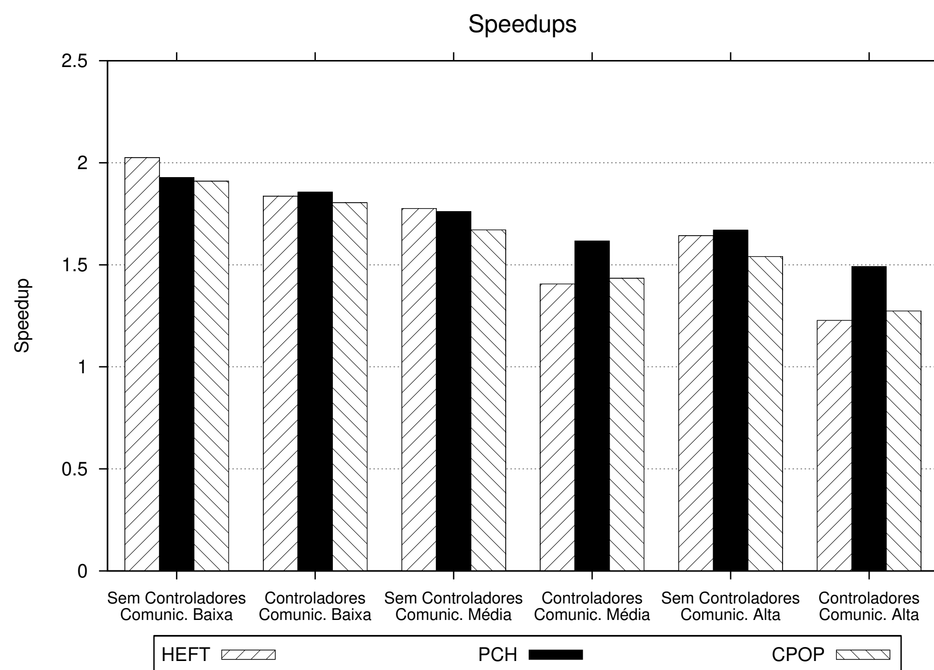

Figura 7.8: *Speedup* médio em no máximo 2 grupos com no máximo 3 recursos.

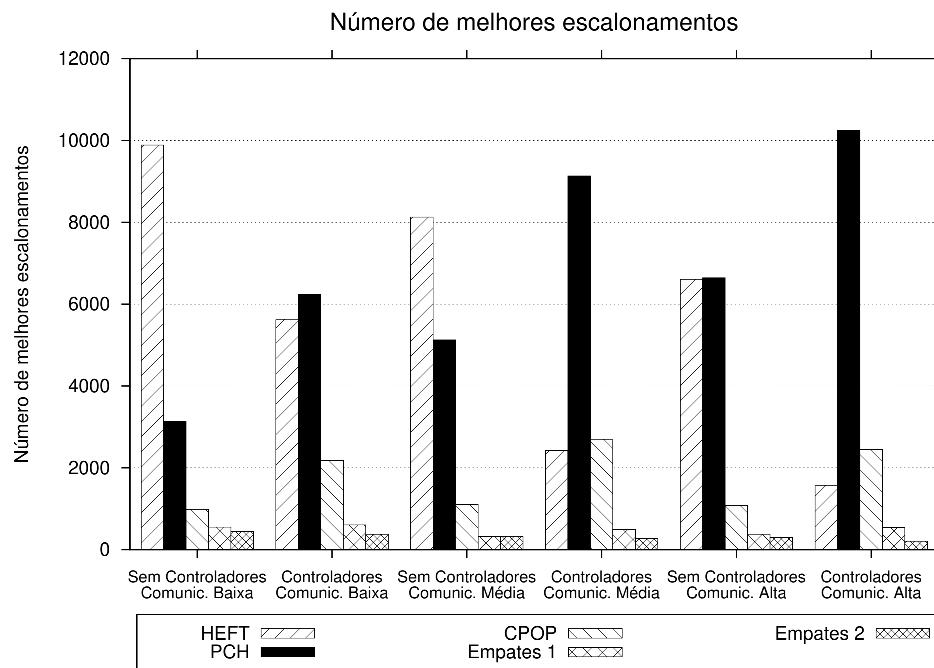

Figura 7.9: Número de escalonamentos com menor *makespan* em no máximo 2 grupos com no máximo 10 recursos.

e 3 recursos cada, há uma pequena melhora no desempenho do PCH em relação ao HEFT e ao CPOP. No caso onde há comunicação baixa e controladores, o PCH alcança o melhor escalonamento em 45.61% das execuções, enquanto com comunicação média esse valor é de 64.17% e com comunicação alta 71.93%.

A Figura 7.10 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 2 grupos com no máximo 10 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.91 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.34 vezes o *overhead* do PCH, quando a comunicação é baixa. Com comunicação média, HEFT tem *overhead* 2.47 vezes o *overhead* gerado pelo PCH, e CPOP tem a mesma taxa anterior, de *overhead* 1.34. Quando a comunicação é alta, a taxa de *overhead* HEFT/PCH é de 2.41 e a taxa de *overhead* CPOP/PCH é de 1.3.

A Figura 7.11 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 2 e número de recursos limitado a 10 por grupo. Os resultados são análogos aos apresentados no cenário com máximo de 2 grupos e 3 recursos.

A Figura 7.12 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 2 grupos com no máximo 10 recursos cada. O PCH apresenta melhores *speedups*

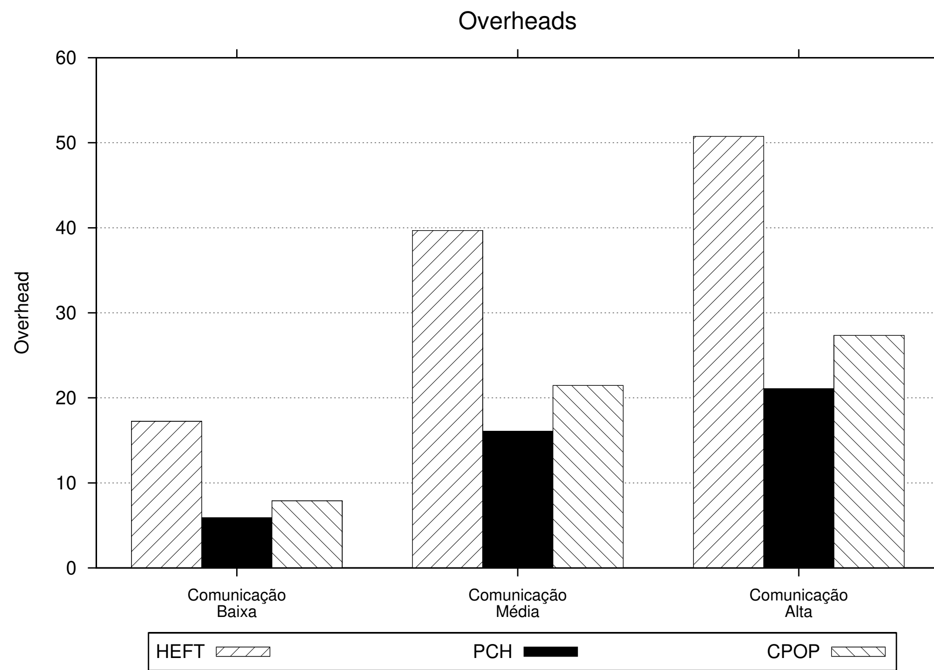

Figura 7.10: *Overhead* de comunicação dos controladores em no máximo 2 grupos com no máximo 10 recursos.

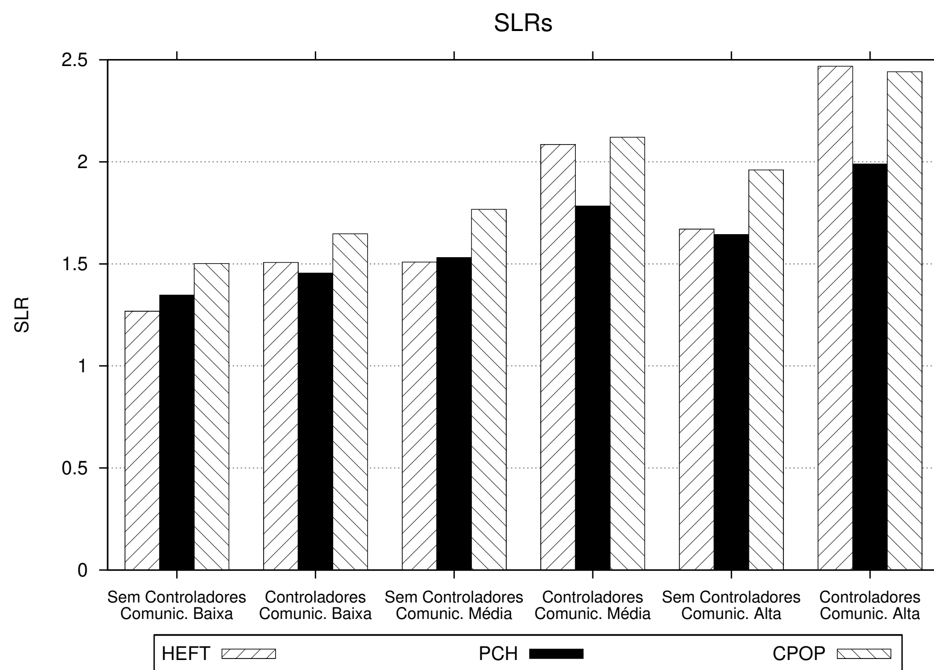

Figura 7.11: SLR médio em no máximo 2 grupos com no máximo 10 recursos.

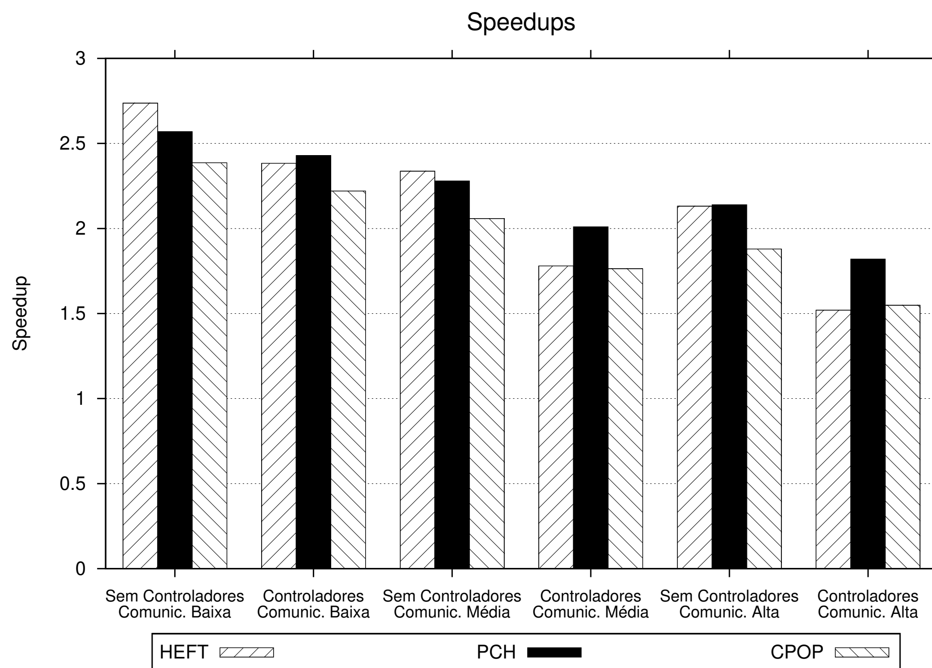

Figura 7.12: *Speedup* médio em no máximo 2 grupos com no máximo 10 recursos.

quando controladores são considerados com qualquer tipo de comunicação.

Os próximos resultados são relativos à execução sob as restrições de no máximo 3 grupos com no máximo 3 recursos cada. A Figura 7.13 mostra o número de vezes que cada algoritmo gerou o melhor escalonamento nesse cenário. O PCH apresentou bom desempenho em todos os casos onde há controladores, registrando o melhor *makespan* em 46.68% das execuções com comunicação baixa, em 73.31% com comunicação média e 77.92% com comunicação alta.

A Figura 7.14 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 3 grupos com no máximo 3 recursos cada. O *overhead* médio nos escalonamentos gerados pelo HEFT com comunicação baixa é da ordem de 3.1 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.53 vezes o *overhead* do PCH. Com comunicação média, esses números são de 2.75 para o HEFT e 1.63 para o CPOP. Para o cenário de comunicação alta, o *overheads* médio dos controladores nas execuções do HEFT é da ordem de 2.62 vezes maior que aquele das execuções do PCH. Nas execuções com CPOP esse valor é 1.56.

A Figura 7.15 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 3 e número de recursos limitado a 3 por grupo. O gráfico confirma o desempenho do PCH superior ao HEFT e CPOP nas execuções com controladores.

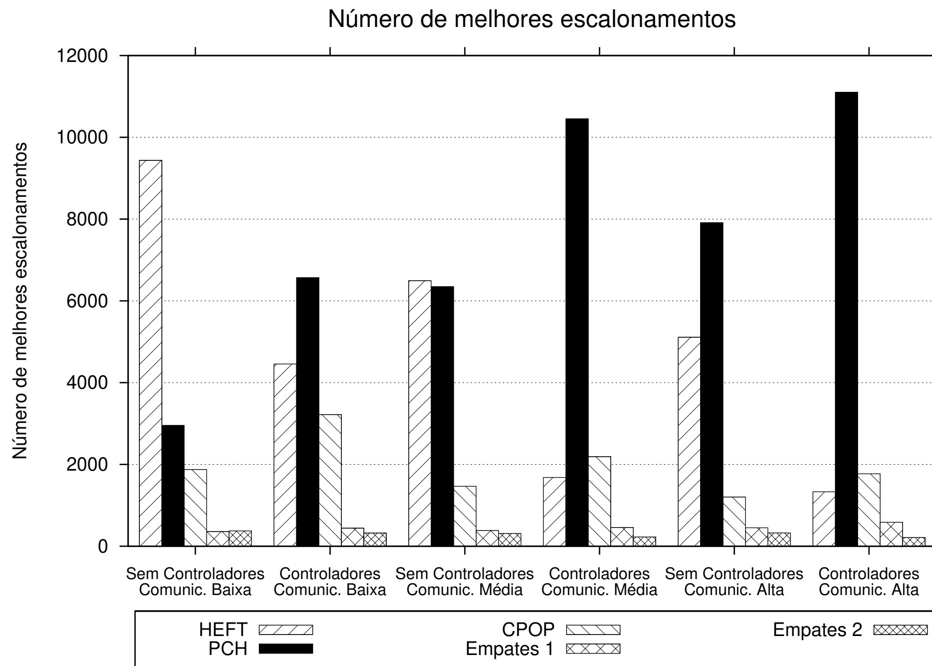

Figura 7.13: Número de escalonamentos com menor *makespan* em no máximo 3 grupos com no máximo 3 recursos.

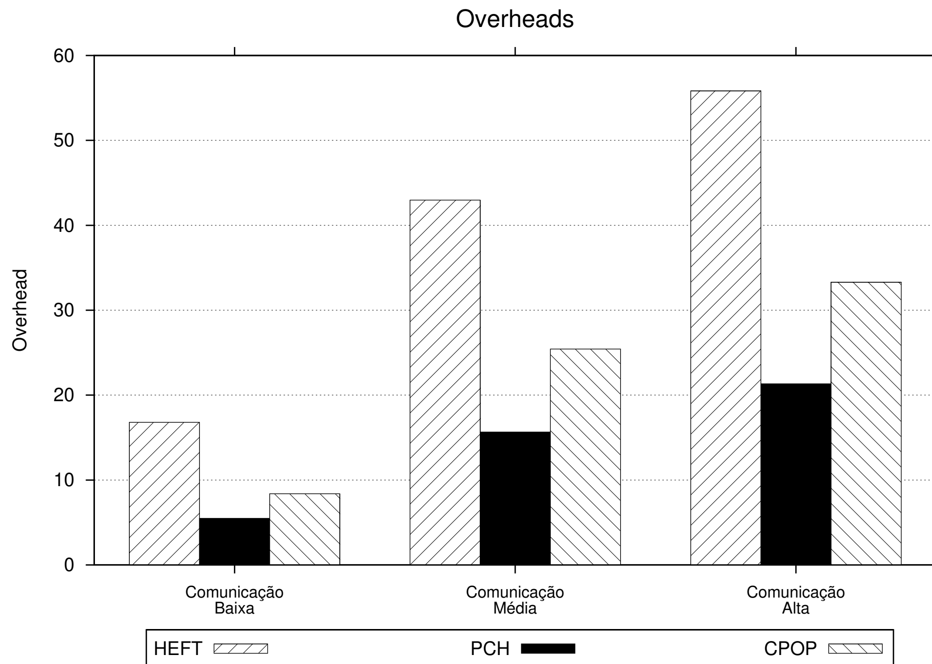

Figura 7.14: *Overhead* de comunicação dos controladores em no máximo 3 grupos com no máximo 3 recursos.

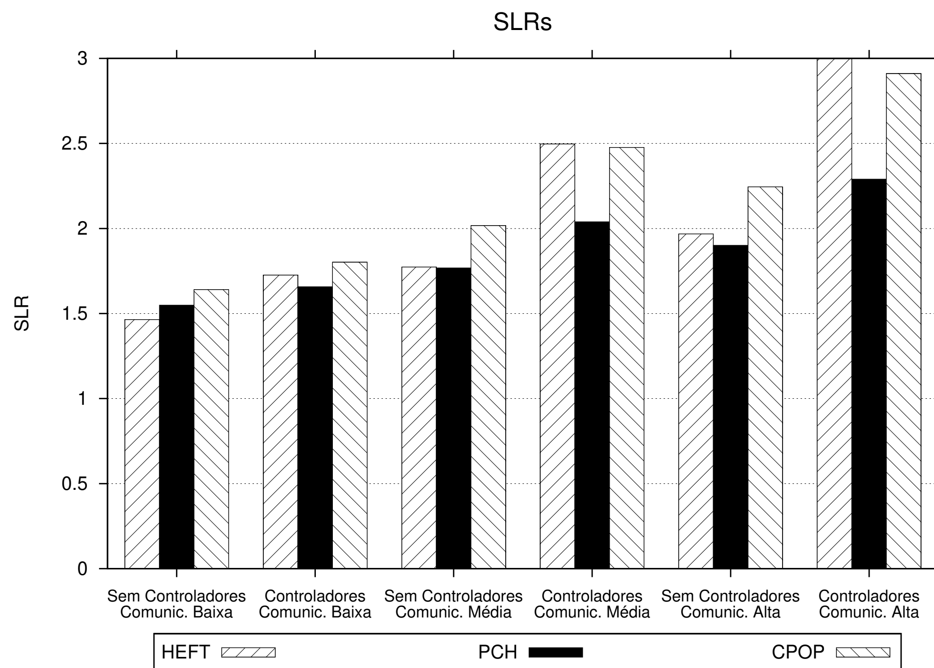

Figura 7.15: SLR médio em no máximo 3 grupos com no máximo 3 recursos.

A Figura 7.16 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 3 grupos com no máximo 3 recursos cada, com o PCH apresentando resultados satisfatórios nas execuções com controladores.

A Figura 7.17 mostra a quantidade de melhores escalonamentos de cada algoritmo quando da execução da simulação em um sistema de no máximo 4 grupos com no máximo 5 recursos cada. O PCH gerou melhores escalonamentos em 53.7% das execuções com controladores e comunicação baixa. Com controladores e comunicação média e alta, o PCH gerou melhores escalonamentos em 74.00% e 79.00% dos casos, respectivamente.

A Figura 7.18 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 4 grupos com no máximo 5 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.7 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.25 vezes o *overhead* do PCH.

A Figura 7.19 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 4 e número de recursos limitado a 5 por grupo. Quando a comunicação é baixa e controladores são considerados, PCH e HEFT têm resultados com diferença ínfima. Com o aumento da comunicação, o PCH mostra desempenho melhor que HEFT e CPOP.

A Figura 7.20 mostra a média dos *speedups* das execuções da cada algoritmo em no

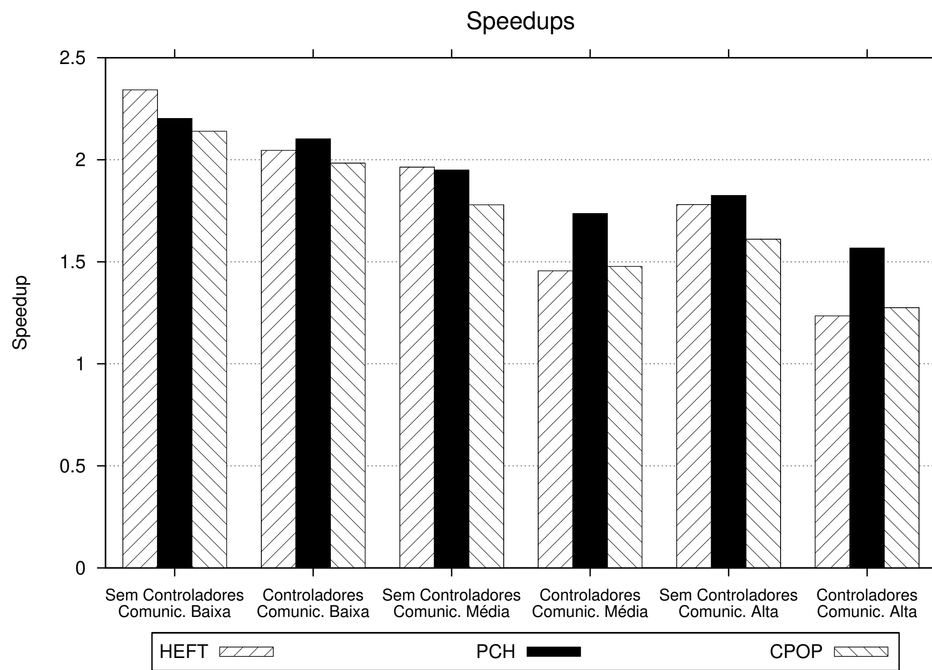

Figura 7.16: *Speedup* médio em no máximo 3 grupos com no máximo 3 recursos.

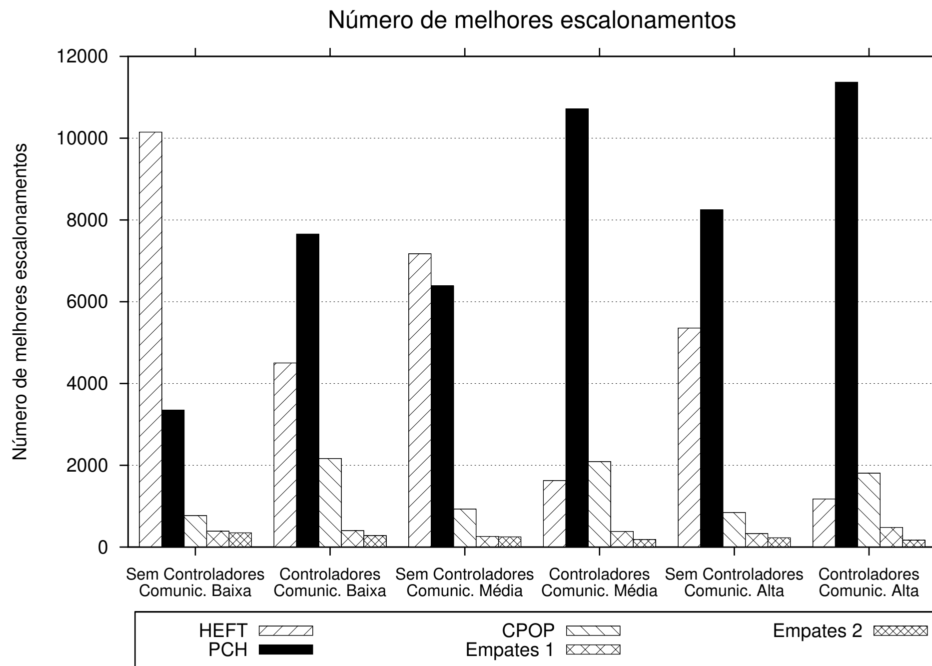

Figura 7.17: Número de escalonamentos com menor *makespan* em no máximo 4 grupos com no máximo 5 recursos.

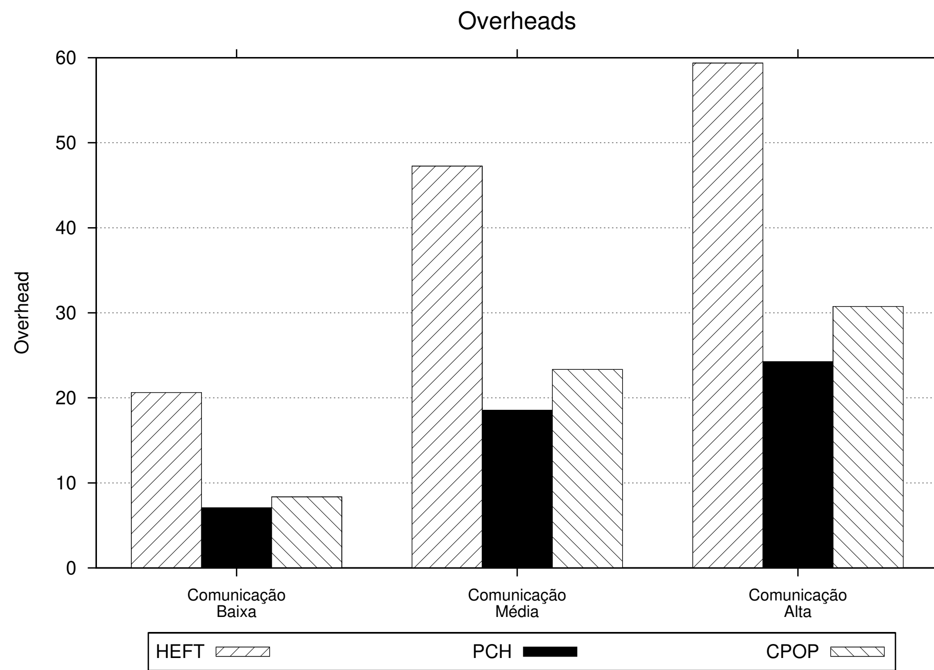

Figura 7.18: *Overhead* de comunicação dos controladores em no máximo 4 grupos com no máximo 5 recursos.

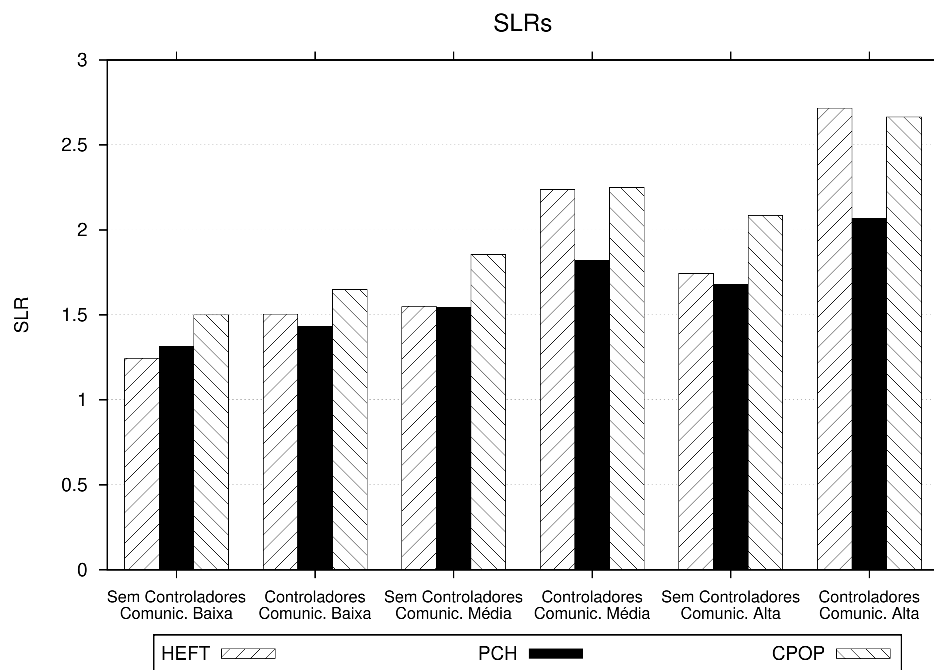

Figura 7.19: SLR médio em no máximo 4 grupos com no máximo 5 recursos.

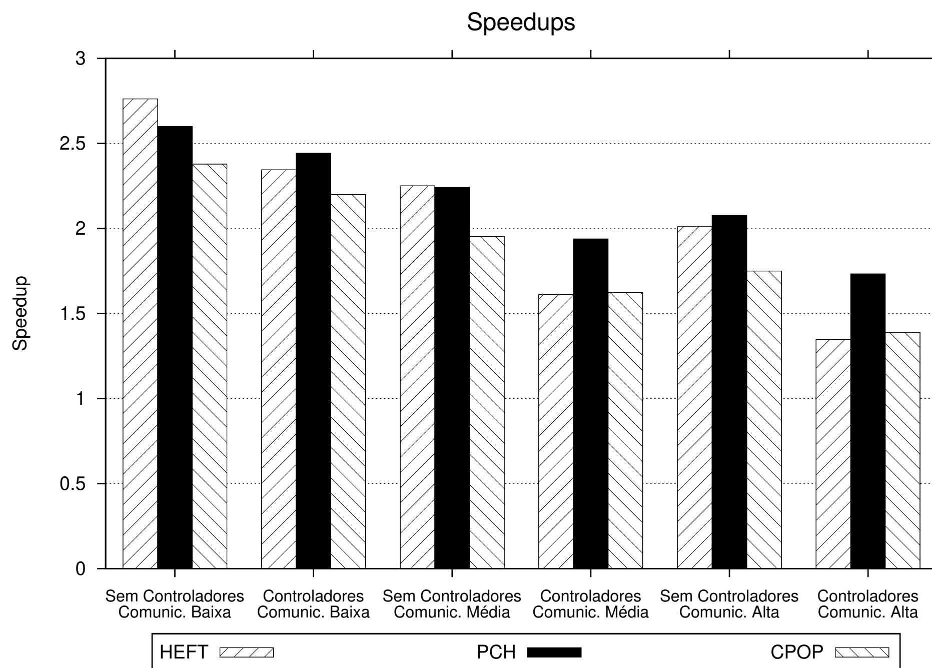

Figura 7.20: *Speedup* médio em no máximo 4 grupos com no máximo 5 recursos.

máximo 4 grupos com no máximo 5 recursos cada. Como no gráfico do SLR (Figura 7.19), PCH e HEFT apresentam resultados similares com comunicação baixa e controladores. Novamente com o aumento da comunicação o PCH se destaca em desempenho.

Os resultados a seguir representam uma simulação com no máximo 5 grupos e no máximo 3 recursos por grupo. A Figura 7.21 mostra a quantidade de melhores escalonamentos de cada algoritmo. O PCH gerou o melhor escalonamento em 54.64% das execuções com controladores e comunicação baixa, resultado similar ao HEFT. Como em todas as simulações, o aumento da comunicação aumenta a vantagem do PCH sobre HEFT e CPOP. Com controladores e comunicação média, PCH alcançou o melhor escalonamento em 77.18% das execuções. Nas execuções com comunicação alta, PCH gerou o melhor *makespan* em 81.72% dos casos.

A Figura 7.22 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 5 grupos com no máximo 3 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.7 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.3 vezes o *overhead* do PCH.

A Figura 7.23 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 5 e número de recursos limitado a 3 por grupo. Com comunicação baixa, média e alta o PCH obtém melhores resultados quando

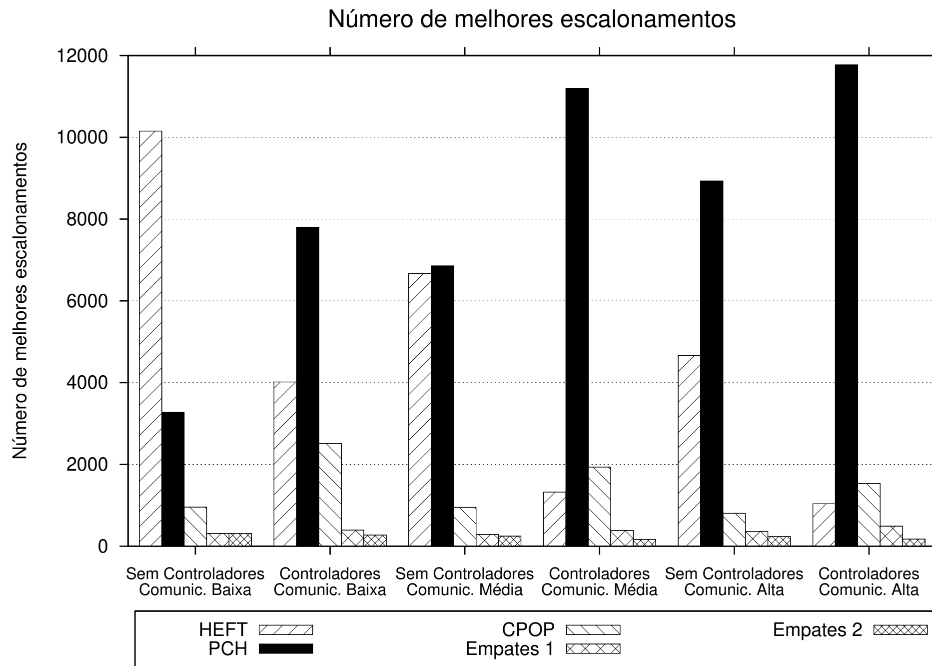

Figura 7.21: Número de escalonamentos com menor *makespan* em no máximo 5 grupos com no máximo 3 recursos.

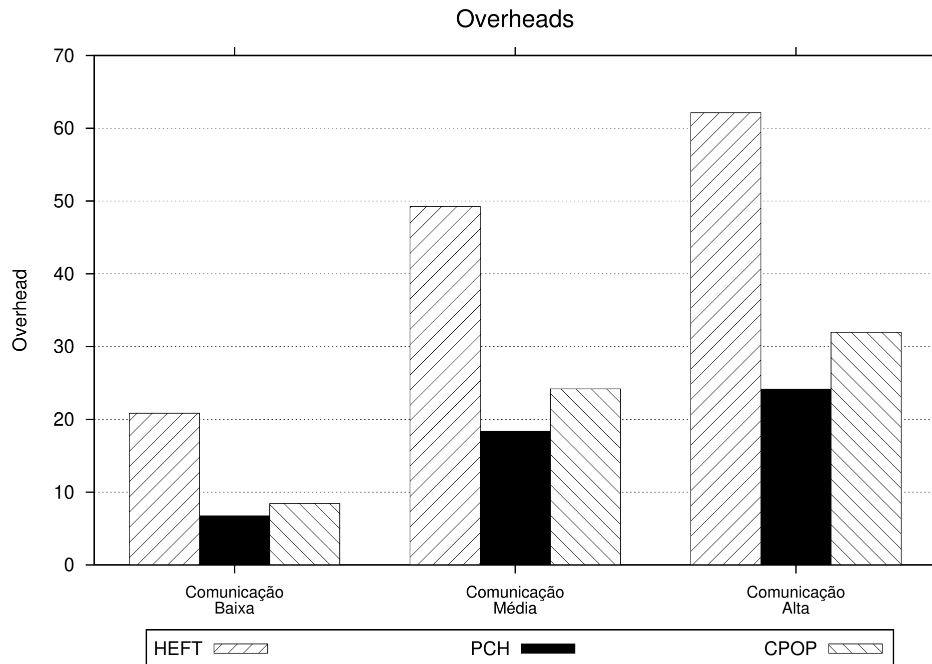

Figura 7.22: *Overhead* de comunicação dos controladores em no máximo 5 grupos com no máximo 3 recursos.

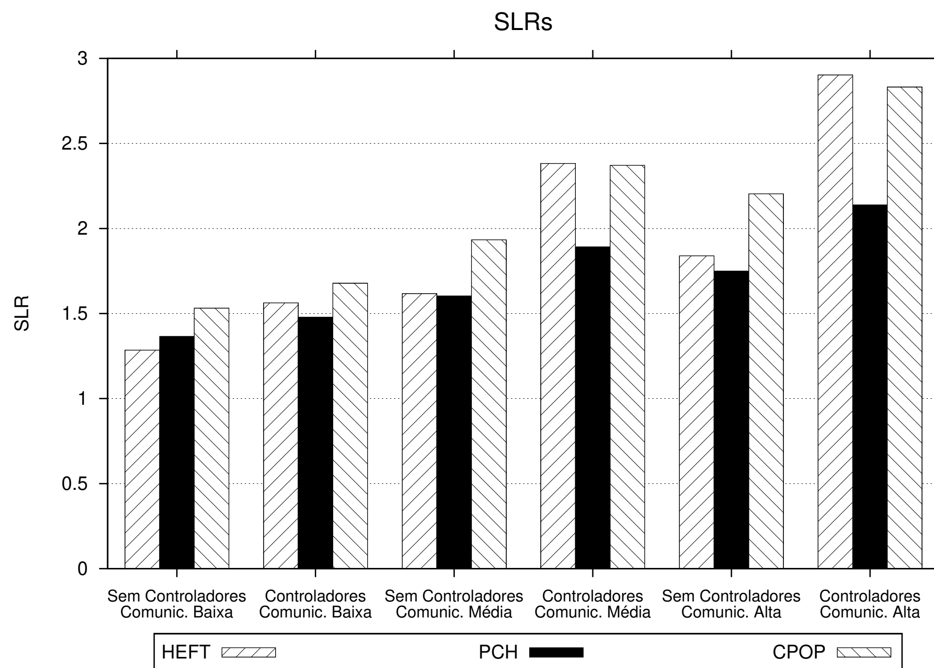

Figura 7.23: SLR médio em no máximo 5 grupos com no máximo 3 recursos.

controladores são considerados, além de ter SLR médio menor nas execuções em que controladores não são considerados e a comunicação é alta.

A Figura 7.24 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 5 grupos com no máximo 3 recursos cada. Novamente, nos quadros de comunicação baixa, média e alta, com controladores, o PCH mantém-se melhor.

A simulação com um limite maior no número de grupos, 8, e com poucos recursos por grupo, no máximo 3, teve como resultados no número de melhores escalonamentos o que mostra a Figura 7.25. Neste cenário o PCH mostra desempenho superior tanto com comunicação baixa (60.69%), quanto com comunicações média (80.00%) e alta (84.12%), considerando controladores.

A Figura 7.26 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 8 grupos com no máximo 3 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.6 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.1 vezes o *overhead* do PCH.

A Figura 7.27 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 8 e número de recursos limitado a 3 por grupo. O PCH apresenta melhores resultados com comunicação baixa, média ou alta, com controladores.

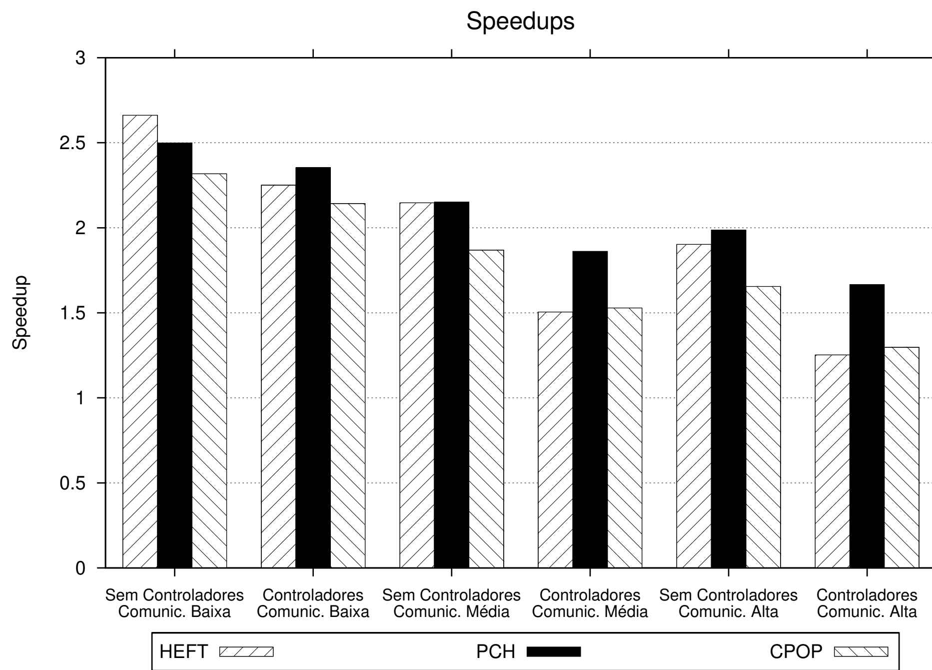

Figura 7.24: *Speedup* médio em no máximo 5 grupos com no máximo 3 recursos.

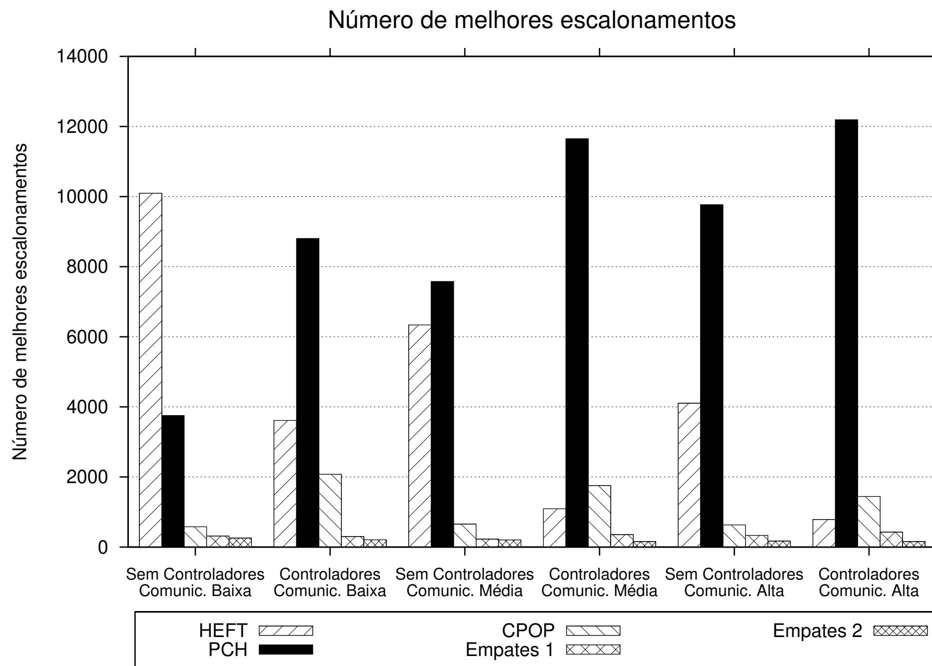

Figura 7.25: Número de escalonamentos com menor *makespan* em no máximo 8 grupos com no máximo 3 recursos.

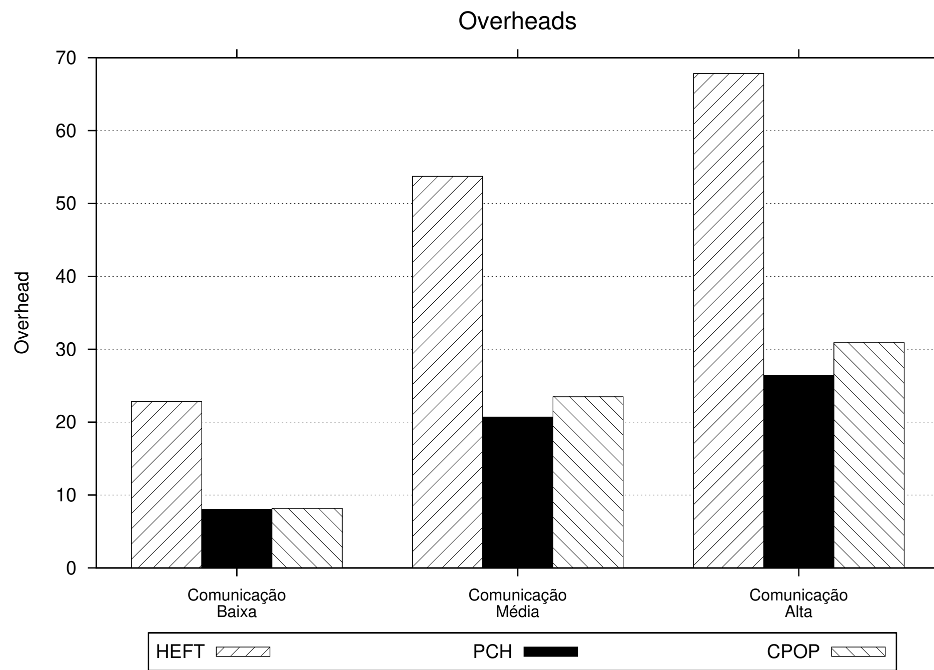

Figura 7.26: *Overhead* de comunicação dos controladores em no máximo 8 grupos com no máximo 3 recursos.

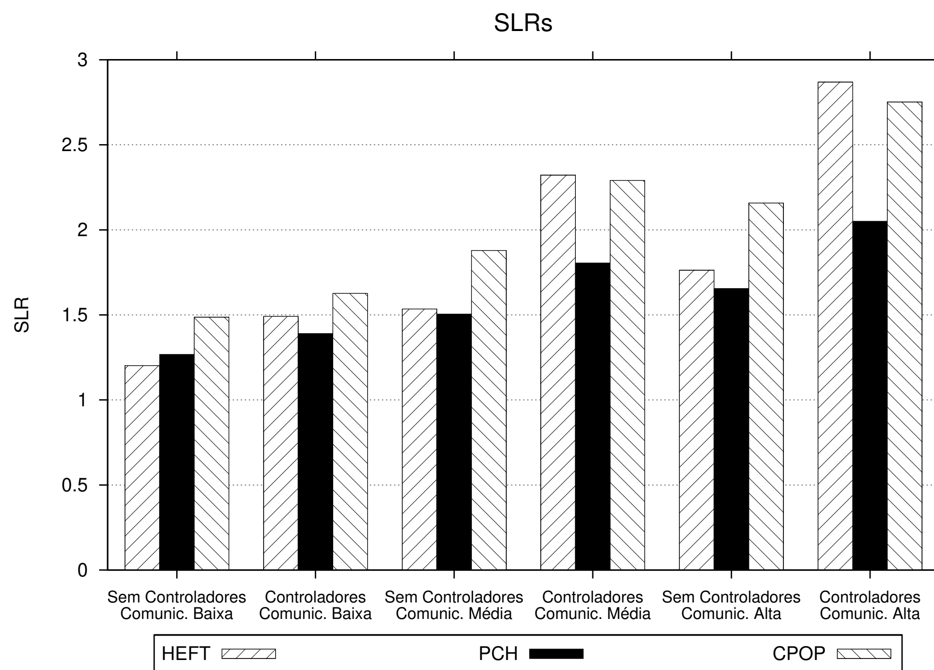

Figura 7.27: SLR médio em no máximo 8 grupos com no máximo 3 recursos.

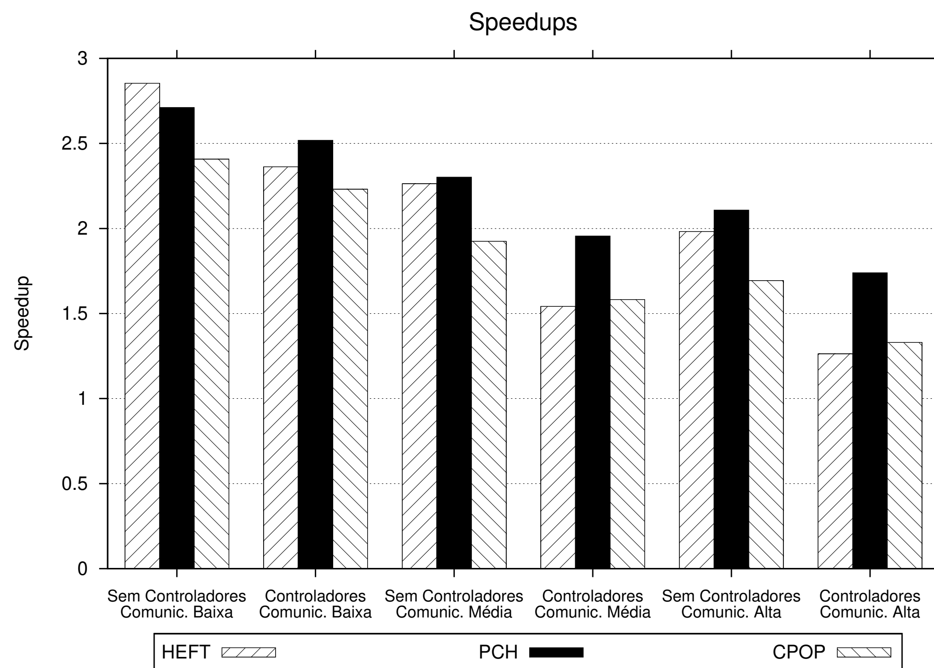

Figura 7.28: *Speedup* médio em no máximo 8 grupos com no máximo 3 recursos.

A Figura 7.28 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 8 grupos com no máximo 3 recursos cada. Os resultados são análogos aos apresentados na figura do SLR médio para as mesmas restrições na quantidade de grupos e recursos (Figura 7.27).

Como última simulação na topologia de grupos, consideramos no máximo 8 grupos com no máximo 10 recursos cada. A Figura 7.29 mostra o número de melhores escalonamentos de cada algoritmo nesse cenário. Como no cenário anterior, o PCH apresenta escalonamentos melhores com controladores e comunicação baixa (60.75%), média (73.78% e alta (78.51%).

A Figura 7.30 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 8 grupos com no máximo 10 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.3 vezes o *overhead* gerado pelo PCH. O *overhead* do CPOP é um pouco menor que o do PCH neste cenário, entretanto os escalonamentos gerados pelo PCH são melhores, como mostram os resultados de SLR e *speedup* para 8 grupos e 10 recursos.

A Figura 7.31 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 8 e número de recursos limitado a 10 por grupo. Em todos os cenários de comunicação com controladores o PCH apresenta desempenho superior ao HEFT e ao CPOP.

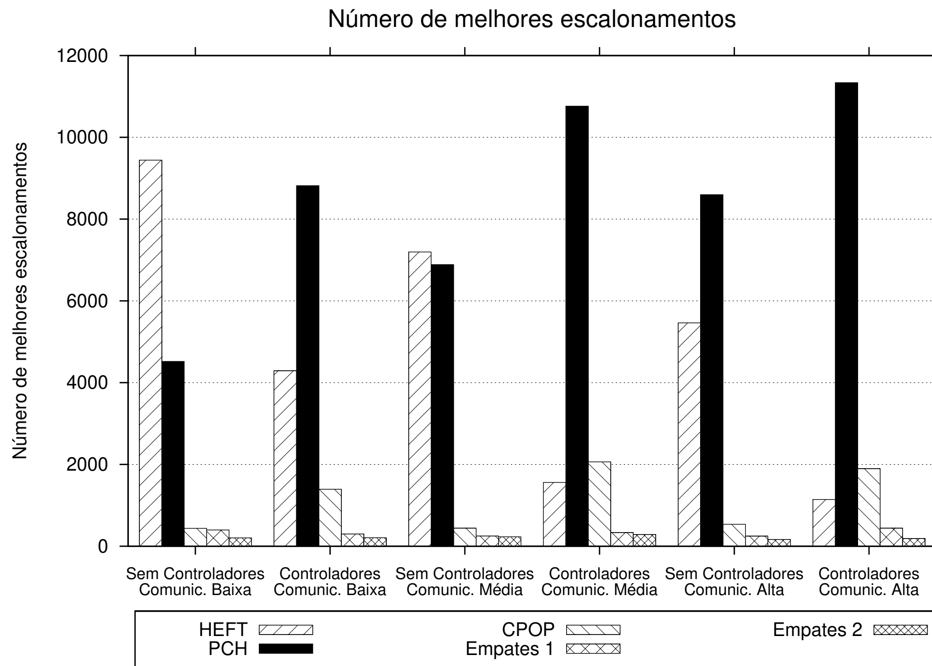

Figura 7.29: Número de escalonamentos com menor *makespan* em no máximo 8 grupos com no máximo 10 recursos.

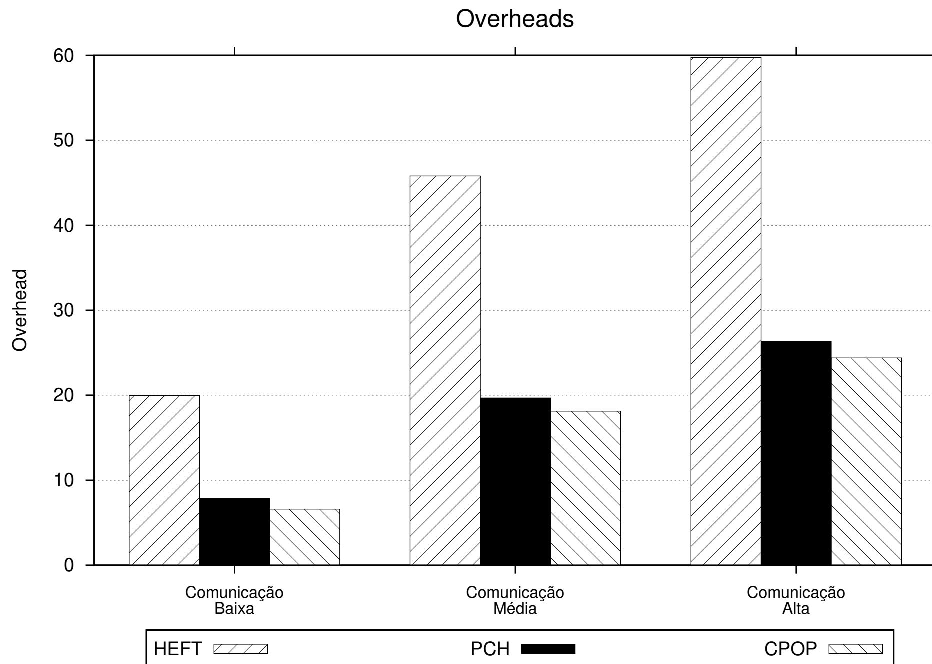

Figura 7.30: *Overhead* de comunicação dos controladores em no máximo 8 grupos com no máximo 10 recursos.

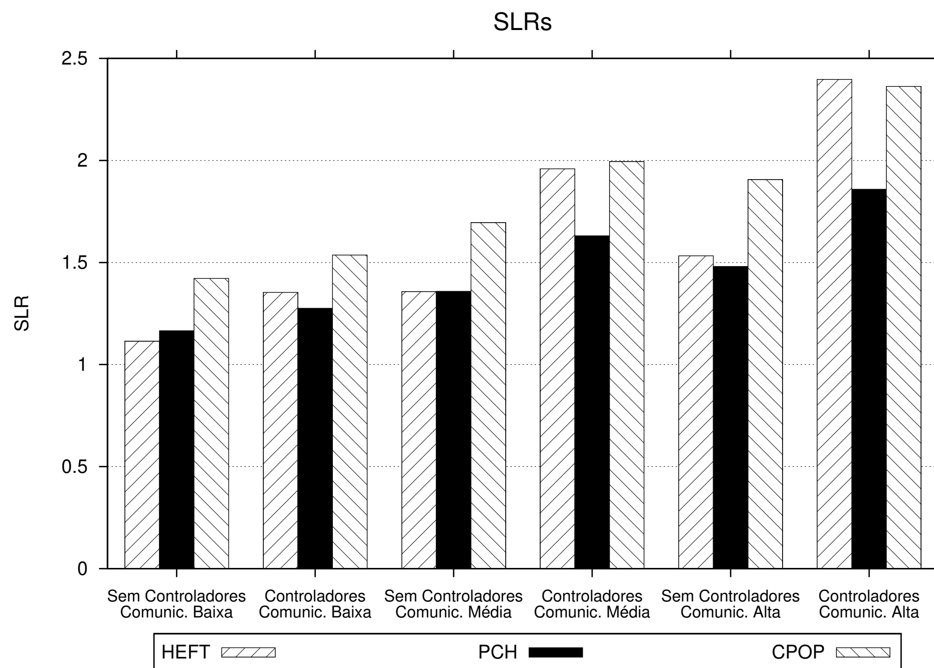

Figura 7.31: SLR médio em no máximo 8 grupos com no máximo 10 recursos.

A Figura 7.32 mostra a média dos *speedups* das execuções da cada algoritmo em no máximo 8 grupos com no máximo 10 recursos cada, confirmando o melhor desempenho do PCH sobre o HEFT e o CPOP nos cenários com controladores.

Os resultados desta seqüência de simulações em topologia de grupos sugerem que o algoritmo PCH é escalável dentro da arquitetura Xavantes. O algoritmo PCH mostra desempenho superior ao CPOP e ao HEFT sempre que controladores são considerados. O PCH apresenta uma melhora de desempenho de acordo com o aumento do número de grupos utilizados. A escalabilidade é um fator importante para qualquer algoritmo com propósito de execução em grades computacionais.

Para complementar os resultados obtidos, as médias e os desvios padrões sobre as simulações em topologias de grupos foram calculados. A evolução dos *makespans* médios sobre todas as execuções com comunicação baixa de acordo com o número de grupos é mostrada na Figura 7.33. Os desvios padrões médios para essas execuções são mostrados na Figura 7.34.

As médias de *makespans* e desvios padrões também são mostradas, em números, na Tabela 7.1. Os desvios padrões são influenciados pelo fato dos custos de computação e comunicação serem gerados aleatoriamente dentro de um intervalo, o que altera aleatoriamente os valores dos *makespans*.

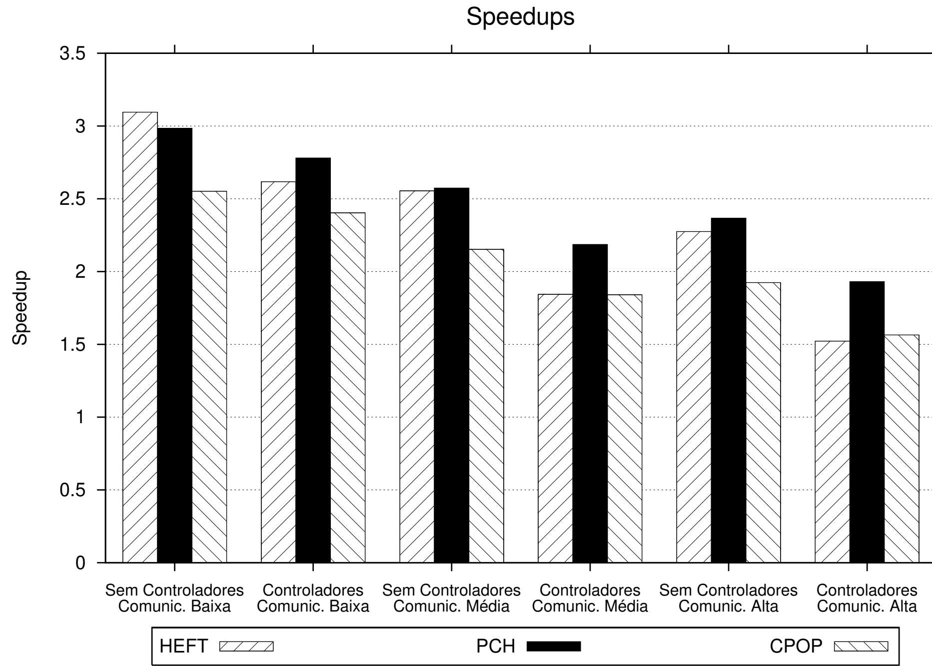

Figura 7.32: *Speedup* médio em no máximo 8 grupos com no máximo 10 recursos.

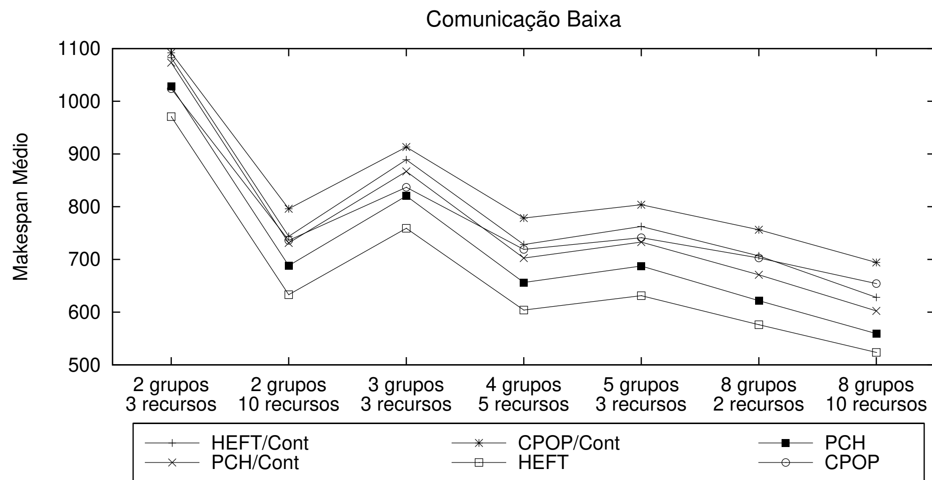

Figura 7.33: *Makespan* médio sobre todas as execuções com comunicação baixa.

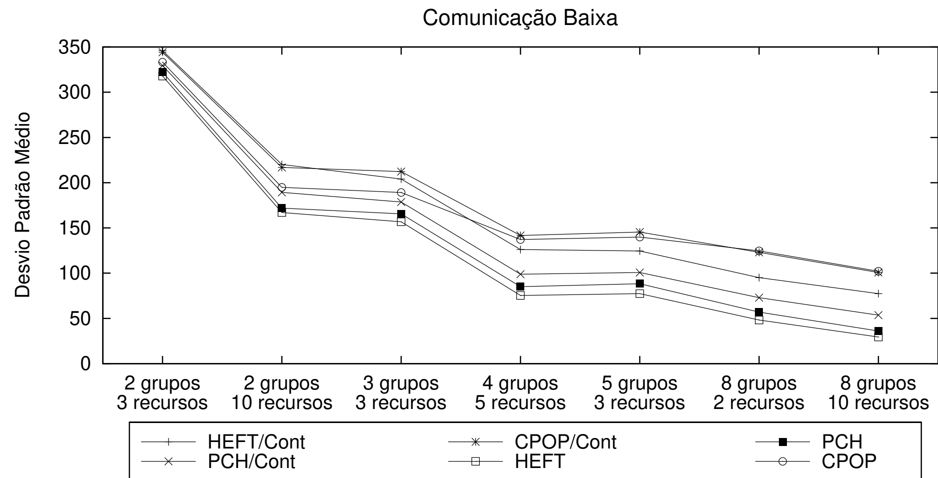

Figura 7.34: Desvio padrão médio para execuções com comunicação baixa.

Tabela 7.1: *Makespans* e desvios padrões médios para execuções com comunicação baixa.

Grupos	Rec.	Contr.	HEFT		PCH		CPOP	
2	3	Não	970.95	317.87	1028.31	322.19	1023.82	333.23
2	3	Sim	1082.69	345.95	1073.05	329.46	1091.98	344.22
2	10	Não	633.05	167.02	688.23	171.90	736.18	194.79
2	10	Sim	743.85	220.14	731.16	189.29	796.047	216.89
3	3	Não	758.92	156.81	820.27	165.60	836.79	189.23
3	3	Sim	889.08	204.05	866.82	178.62	913.26	212.33
4	5	Não	603.92	75.36	656.19	85.05	718.92	137.22
4	5	Sim	728.14	126.15	702.63	98.85	778.44	141.70
5	3	Não	631.21	77.39	687.60	88.36	741.28	139.94
5	3	Sim	762.61	124.54	732.95	100.71	803.81	145.39
8	3	Não	576.10	48.16	621.77	56.97	702.69	124.60
8	3	Sim	706.82	95.06	670.90	72.86	756.32	123.07
8	10	Não	523.61	29.41	559.16	36.04	654.18	102.04
8	10	Sim	628.09	77.41	602.35	53.55	694.12	100.77

As médias de *makespan* e desvios padrões em cada cenário de grupos, com comunicação média, são mostradas na Figura 7.35 e na Figura 7.36, respectivamente.

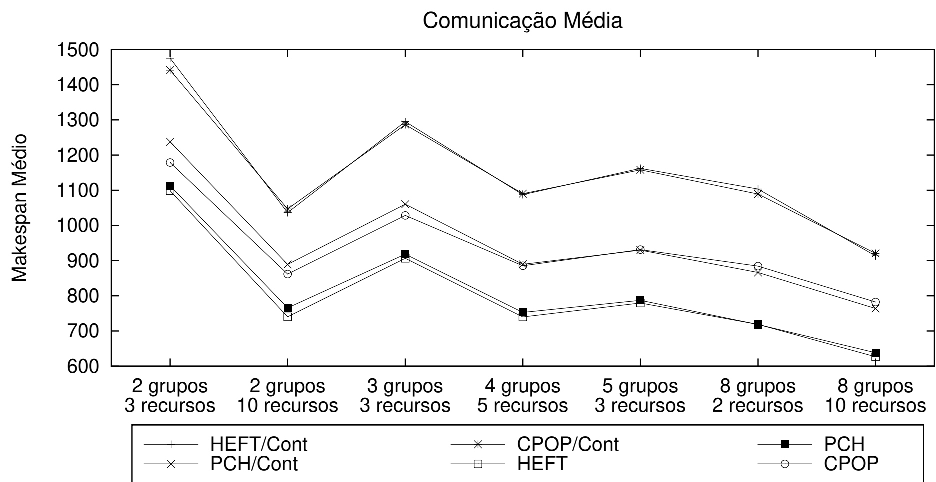

Figura 7.35: *Makespan* médio sobre todas as execuções com comunicação média.

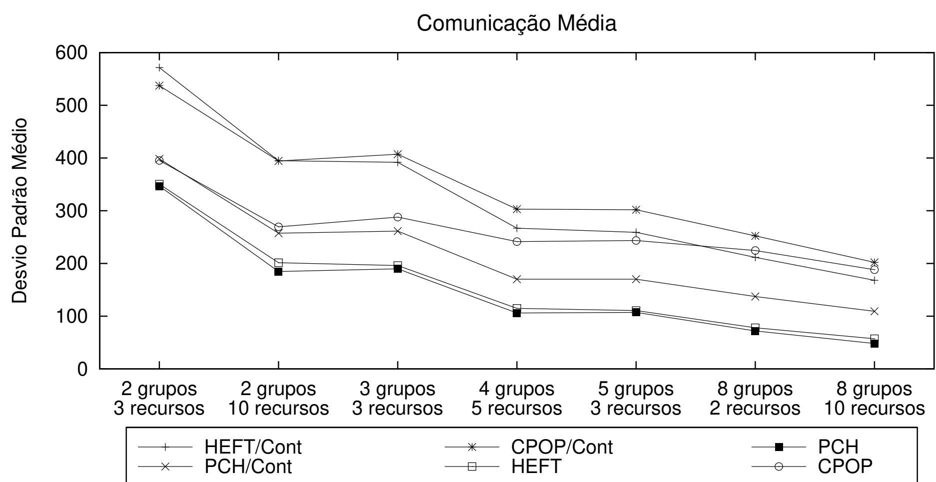

Figura 7.36: Desvio padrão médio para execuções com comunicação média.

A Tabela 7.2 mostra os valores numéricos das médias e desvios padrões para execuções com comunicação média.

Tabela 7.2: *Makespans* e desvios padrões médios para execuções com comunicação média.

Grupos	Rec.	Contr.	HEFT		PCH		CPOP	
2	3	Não	1098.72	350.35	1112.78	395.41	1178.85	395.41
2	3	Sim	1475.46	571.56	1237.98	398.49	1441.33	537.36
2	10	Não	740.15	201.49	765.62	184.73	861.89	269.47
2	10	Sim	1036.34	394.88	889.12	257.43	1046.88	394.51
3	3	Não	906.63	196.14	918.25	189.74	1028.49	287.94
3	3	Sim	1295.27	391.91	1060.30	261.34	1286.36	407.07
4	5	Não	739.80	114.77	752.84	105.93	885.35	241.47
4	5	Sim	1087.63	266.81	889.57	170.18	1090.23	302.97
5	3	Não	779.38	110.67	787.11	107.18	931.34	243.63
5	3	Sim	1162.25	258.95	929.72	170.08	1158.05	301.91
8	3	Não	718.75	78.06	718.55	72.05	884.27	224.64
8	3	Sim	1103.36	211.61	866.15	137.25	1089.38	252.29
8	10	Não	626.75	57.13	638.36	48.17	782.32	188.20
8	10	Sim	913.46	168.05	763.88	109.36	920.41	202.07

Com comunicação alta, as médias de *makespans* em cada cenário de grupos são mostradas na Figura 7.37.

A Figura 7.38 mostra a evolução da média dos desvios padrões de todas as execuções de acordo com a quantidade de grupos e recursos. Os valores numéricos para as médias de *makespans* e desvios padrões são mostrados na Tabela 7.3.

Nas execuções com controladores, o PCH obtém melhor média de *makespan* e de desvios padrões em todos os casos considerados, com comunicação baixa, média ou alta. A média mais baixa com o desvio padrão mais baixo mostra que a maioria dos escalonamentos gerados pelo PCH não tem *makespans* muito grandes quando comparados com os *makespans* gerados pelo HEFT e pelo CPOP. Também é um indício de que HEFT e CPOP geram muitos escalonamentos com *makespans* acima da faixa de variação daqueles gerados pelo PCH, e poucos escalonamentos com *makespans* baixos, abaixo da faixa de variação daqueles gerados pelo PCH.

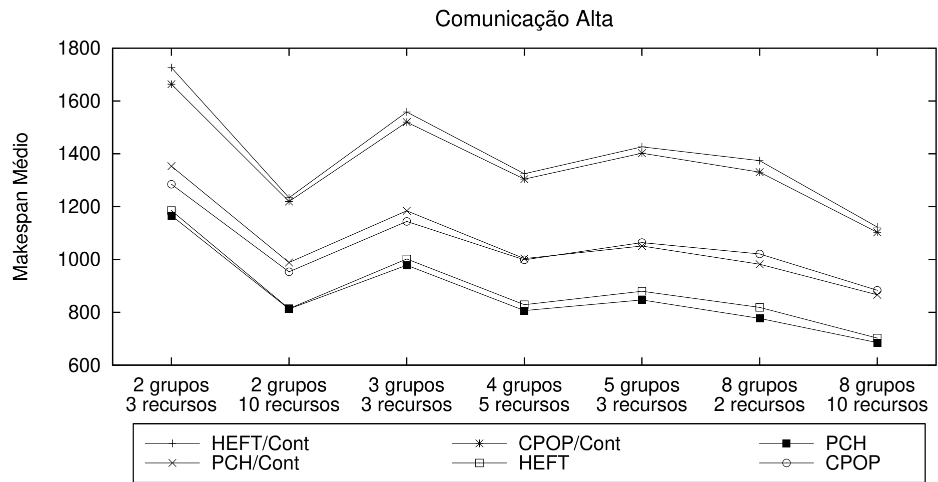

Figura 7.37: *Makespan* médio sobre todas as execuções com comunicação alta.

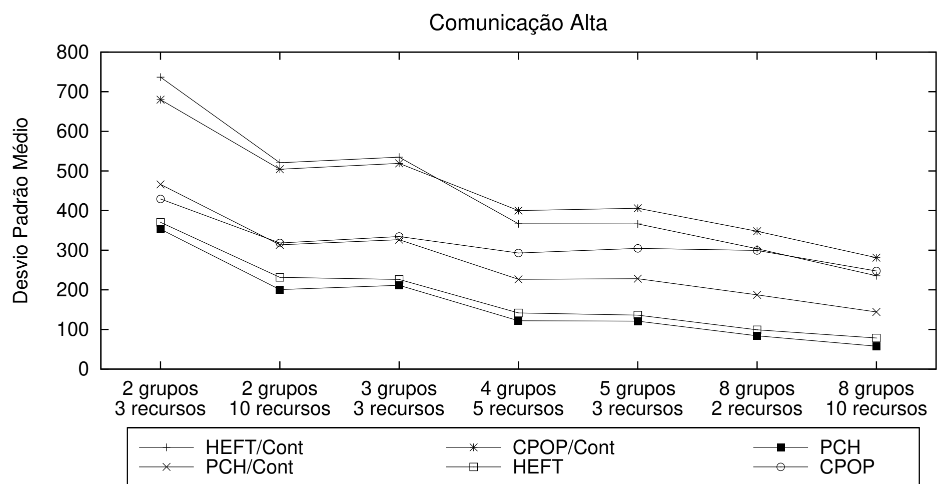

Figura 7.38: Desvio padrão médio para execuções com comunicação alta.

Tabela 7.3: *Makespans* e desvios padrões médios para execuções com comunicação alta.

Grupos	Rec.	Contr.	HEFT		PCH		CPOP	
2	3	Não	1185.37	370.42	1166.28	353.07	1284.72	429.45
2	3	Sim	1726.55	736.64	1353.21	466.16	1663.62	679.93
2	10	Não	813.84	231.44	812.32	200.66	953.22	318.27
2	10	Sim	1233.71	520.76	988.54	313.55	1219.24	504.24
3	3	Não	1001.86	226.15	978.45	211.28	1144.02	334.44
3	3	Sim	1558.31	534.80	1184.12	326.57	1519.88	519.22
4	5	Não	828.56	141.79	806.49	122.057	998.62	292.98
4	5	Sim	1324.45	366.66	1003.02	226.69	1304.20	399.95
5	3	Não	879.78	136.146	846.95	121.00	1063.97	304.62
5	3	Sim	1426.47	366.44	1051.03	228.19	1402.24	405.74
8	3	Não	818.47	99.22	776.97	84.08	1020.70	299.41
8	3	Sim	1374.01	303.76	981.77	187.42	1330.41	347.84
8	10	Não	702.37	78.28	685.28	57.63	883.40	247.63
8	10	Sim	1122.89	235.63	866.21	144.24	1102.85	281.10

7.4.3 Busca de Recursos em Grupos Adjacentes

Nos experimentos apresentados até aqui os algoritmos consideram apenas os recursos que estão no repositório no momento do início do escalonamento de um processo. Nesta seção introduzimos no PCH a estratégia de busca de recursos em grupos adjacentes apresentada na Seção 6.6, resultando em melhoras nos *makespans* dos processos escalonados. Os algoritmos HEFT e CPOP não possuem tal adicional, pois trata-se de uma estratégia desenvolvida para o algoritmo PCH, atuando dentro do middleware Xavantes. Portanto, HEFT e CPOP continuam realizando o escalonamento com os recursos disponíveis no repositório no momento do início do escalonamento de cada processo. A topologia de grupos e as variáveis aleatórias são as mesmas utilizadas no experimento anterior.

Uma questão na busca de recursos em grupos adjacentes é quantos recursos serão retornados pelo grupo que recebe a requisição. Para os experimentos desta seção definimos que cada requisição pode receber como resposta informações sobre 0, 1, 2 ou 3 recursos adicionais, com probabilidades $\frac{2}{3}$, $\frac{1}{9}$, $\frac{1}{9}$ e $\frac{1}{9}$, respectivamente. Como no escalonamento de um processo pode ser feita mais de uma requisição, 0 recursos retornados significa que o grupo já tem informações sobre todos os recursos do grupo adjacente. Adicionalmente, os recursos retornados podem ter capacidade de processamento igual ou menor àquela do melhor recurso que pertence ao grupo alvo da solicitação e que está disponível no repositório do grupo solicitante.

A Figura 7.17 mostra a quantidade de melhores escalonamentos de cada algoritmo quando da execução da simulação em um sistema de no máximo 4 grupos com no máximo 5 recursos cada, com o PCH realizando busca de recursos em grupos adjacentes. O PCH gerou melhores escalonamentos em 65.9% das execuções com controladores e comunicação baixa. Com controladores e comunicação média e alta, o PCH gerou melhores escalonamentos em 76.4% e 78.24% dos casos, respectivamente. Em relação aos resultados anteriores, com no máximo 4 grupos de no máximo 5 recursos e sem busca de recursos em grupos adjacentes, o PCH mostra uma perceptível melhora no desempenho quando a comunicação é baixa, leve melhora com comunicação média e estabilidade com comunicação alta, com controladores. Sem controladores a melhora é substancial em todos os cenários de comunicação.

A Figura 7.40 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores com no máximo 4 grupos com no máximo 5 recursos cada. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 2.6 vezes o *overhead* gerado pelo PCH, enquanto o *overhead* do CPOP é da ordem de 1.4 vezes o *overhead* do PCH.

A Figura 7.41 mostra o SLR médio das execuções de cada algoritmo na topologia de grupos, com número de grupos limitado a 4 e número de recursos limitado a 5 por grupo e com o PCH realizando busca de recursos quando necessário. Com controladores

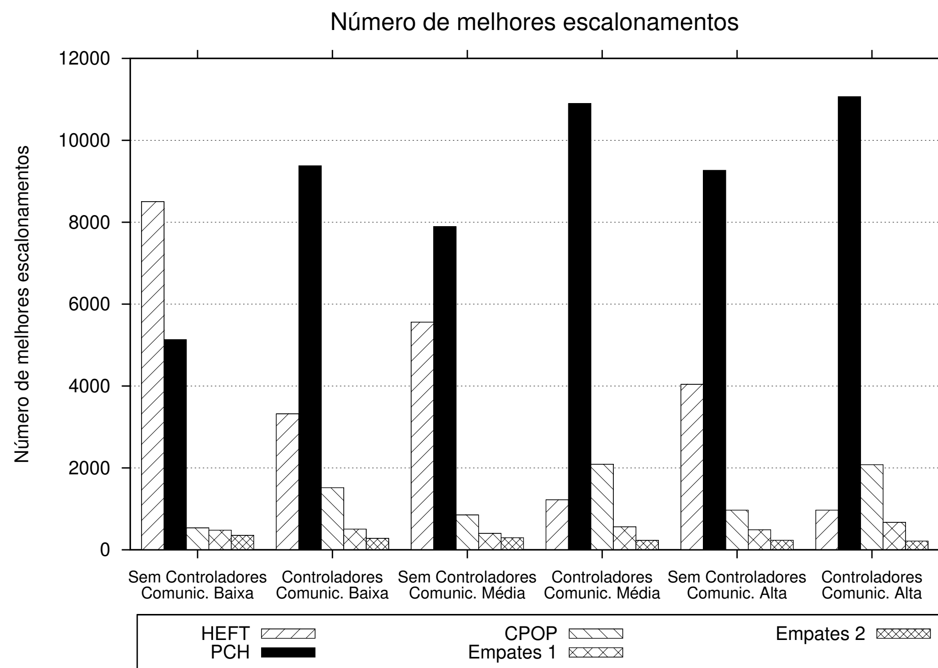

Figura 7.39: Número de escalonamentos com menor *makespan* em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos.

e comunicação baixa, média e alta, o PCH mostrou melhor desempenho que HEFT e CPOP, com resultados melhores que aqueles apresentados sem a busca de recursos em grupos adjacentes. Com comunicação baixa e sem controladores, o PCH mostrou desempenho equivalente ao HEFT, superando o CPOP. Com comunicação média e alta, sem controladores, o PCH mostrou o melhor desempenho entre os três algoritmos.

A Figura 7.42 mostra a média dos *speedups* das execuções da cada algoritmo, com busca de recursos e no máximo 4 grupos com no máximo 5 recursos cada. Como no gráfico do SLR (Figura 7.41), o PCH é mais eficiente com controladores em qualquer cenário de comunicação e sem controladores com comunicação média e alta.

Os resultados apresentados nesta seção mostram que a busca de recursos em grupos adjacentes inserida no algoritmo PCH, possível devido à estrutura de grupos do middleware Xavantes, tem como conseqüência a melhora dos resultados nos escalonamentos realizados em grupos de recursos.

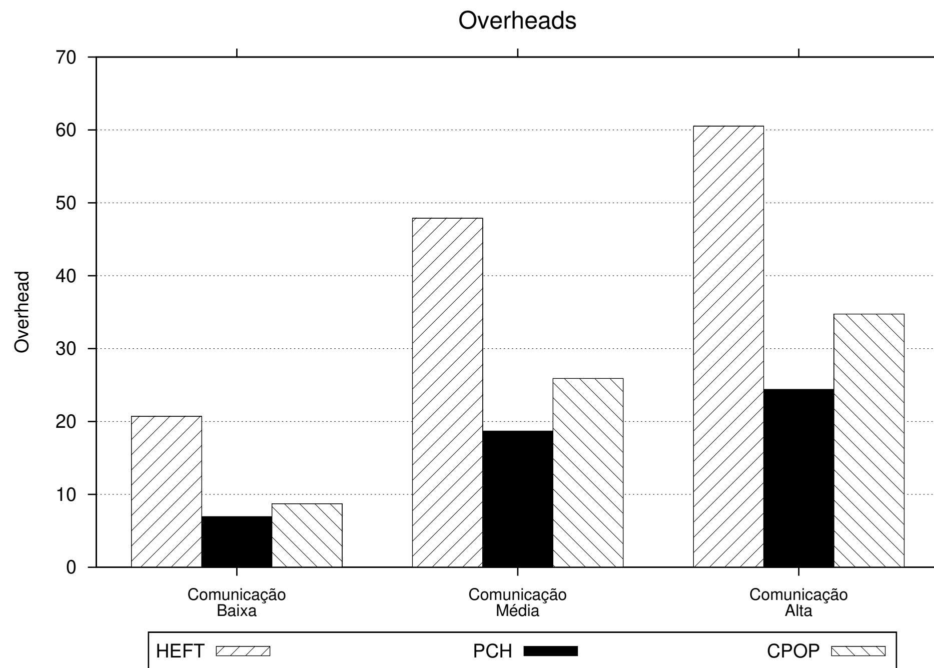

Figura 7.40: *Overhead* de comunicação dos controladores em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos.

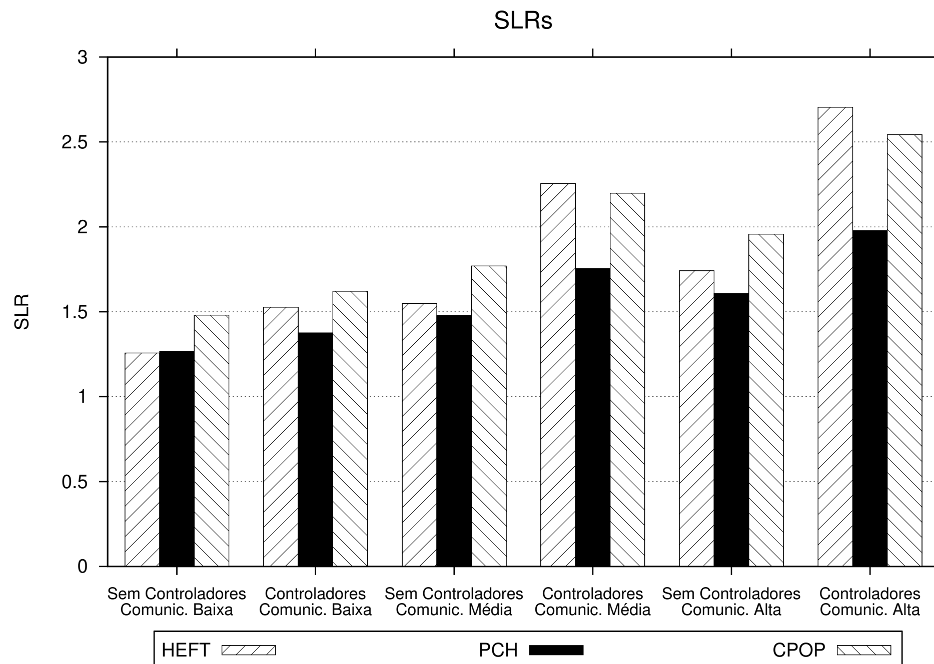

Figura 7.41: SLR médio em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos.

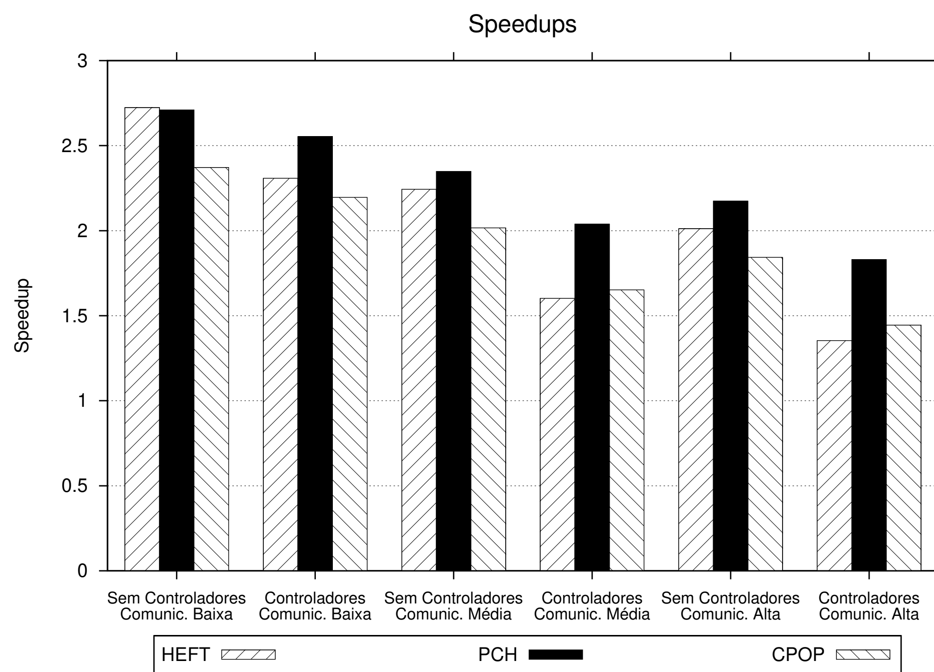

Figura 7.42: *Speedup* médio em no máximo 4 grupos com no máximo 5 recursos, com busca de recursos.

7.4.4 Recursos de Grupos Adjacentes no Repositório Local

Cada grupo tem em seu repositório infomações sobre 20% dos melhores recursos de cada grupo adjacente. Simulações foram realizadas com o intuito de quantificar a influência de tal porcentagem no escalonamento dos processos. Nas simulações desta seção, o número de grupos é variável e o número de recursos por grupo é um valor aleatório entre 1 e 10. Comparamos os resultados de simulações com valores de 10%, 20%, 40% e 60%. A Figura 7.43 mostra a evolução da quantidade de melhores escalonamentos para cada porcentagem considerada. O algoritmo PCH tem melhor desempenho em relação ao HEFT e ao CPOP de acordo com o aumento no número de grupos em qualquer porcentagem.

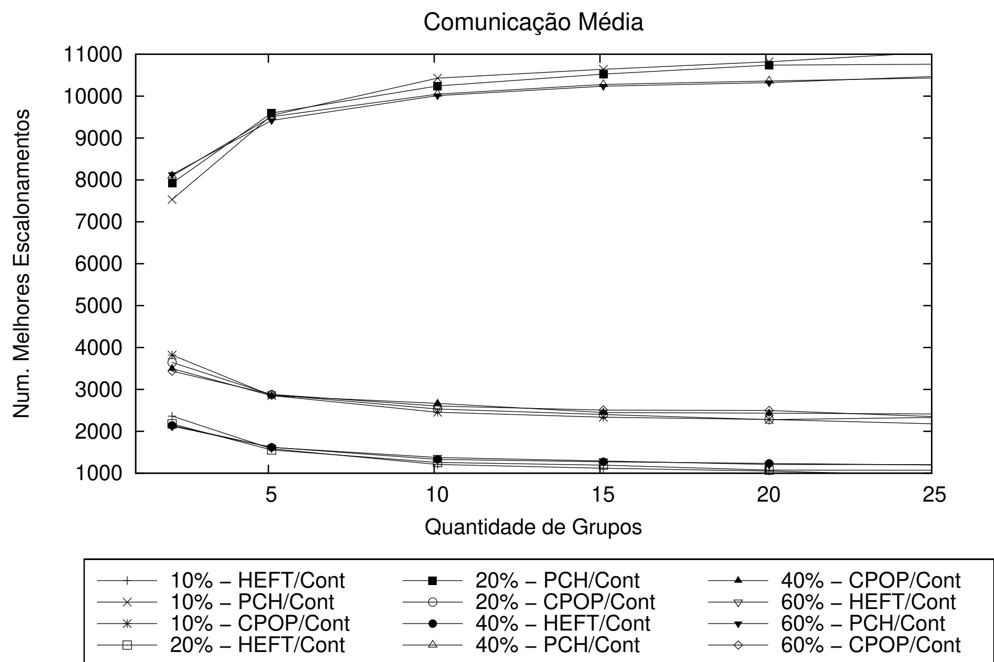

Figura 7.43: Número de melhores escalonamentos de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo.

As Figuras 7.44 e 7.45 mostram a evolução do SLR médio e *speedup* médio com a variação no número de grupos e na porcentagem de recursos de grupos adjacentes conhecida. Como esperado, com o aumento da porcentagem de recursos conhecidos, melhora o desempenho dos algoritmos, pois existem mais recursos disponíveis no repositório local. Nas execuções do PCH há uma visível melhora de desempenho entre 10% e 20%. Entre 20% e 40% a diferença de desempenho é equivalente àquela entre 10% e 20%, porém a quantidade de recursos no repositório é substancialmente maior. A diferença de desempenho quando consideramos 40% e 60% é menor, o que sugere que se um grupo conhece 60% dos recursos de cada um de seus grupos adjacentes, os recursos externos disponíveis aca-

bam não sendo totalmente utilizados no escalonamento. Note que ao aumentar o número de recursos no repositório há também um aumento no tempo de execução do algoritmo. Se considerarmos que o número de grupos e recursos pode ser muito grande na grade, o número de recursos no repositório pode ser também muito grande, tornando excessivo o tempo de execução do algoritmo. Simulações mais detalhadas devem ser realizadas para determinar o balanceamento ideal entre número de recursos externos e tempo de execução.

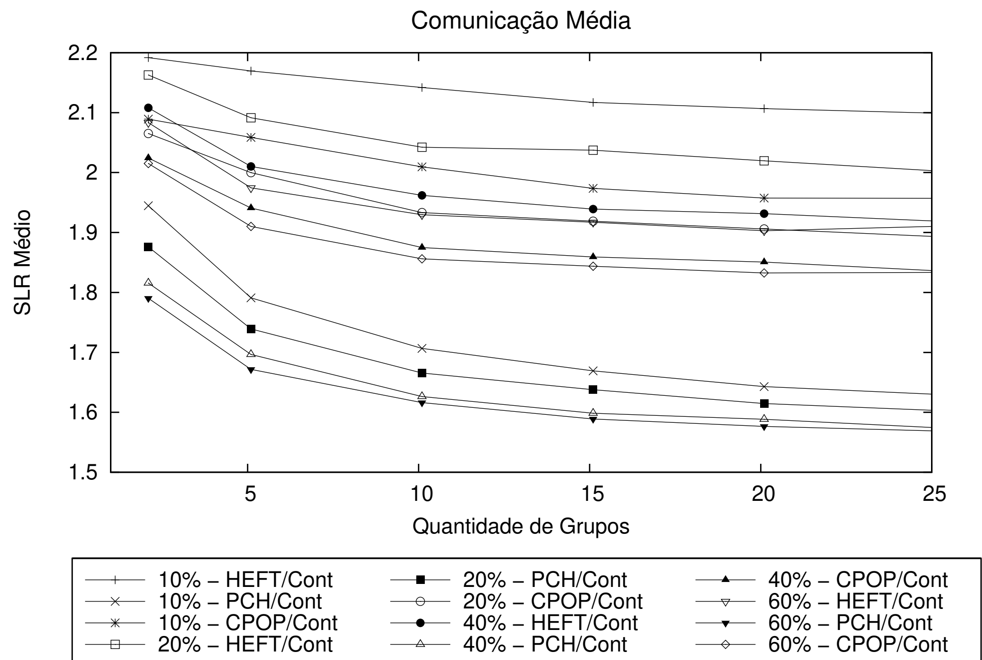

Figura 7.44: SLR médio de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo.

As simulações apresentadas nesta seção mostram que há um limite onde o número de recursos externos disponíveis no repositório local proporciona melhora de desempenho. Após tal limite há uma quantidade demasiada de recursos disponíveis, o que pode causar atraso no escalonamento dos processos. O valor de 20% utilizado no Xavantes está abaixo desse limite, aliando bom desempenho à capacidade de realização de uma busca de recursos em grupos adjacentes que não ocasiona o aumento desmedido do tamanho do repositório local.

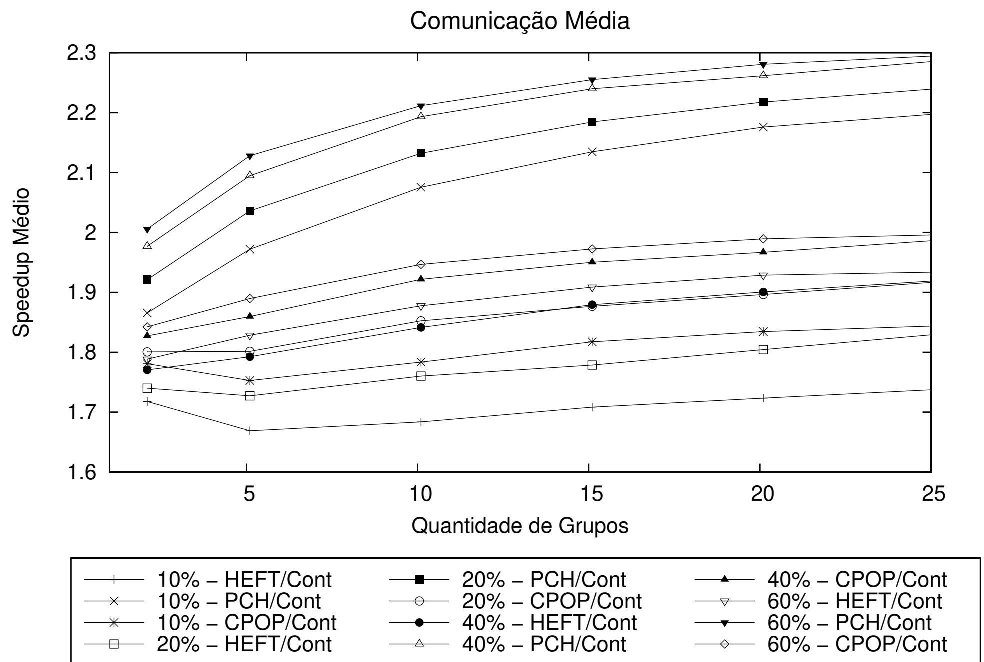

Figura 7.45: *Speedup* médio de acordo com o número de recursos de grupos adjacentes no repositório de cada grupo.

Capítulo 8

Conclusão

Esta dissertação apresenta uma heurística para escalonamento de tarefas no Xavantes [3], um middleware para grades computacionais desenvolvido para a execução de processos tipo workflow. O algoritmo aqui proposto, chamado de *Path Clustering Heuristic* (PCH), pode ser utilizado para escalonamento de tarefas tanto em sistemas heterogêneos como homogêneos. Entretanto, seu objetivo é o escalonamento no middleware Xavantes considerando controladores, fornecendo alta disponibilidade, escalabilidade, tolerância a falhas e recuperação, confiabilidade e desempenho.

As simulações realizadas mostram que, para os grafos de tarefas válidos no Xavantes, a estratégia de construir *clusters* de tarefas baseados em caminhos do grafo proporciona bom desempenho quando controladores são considerados e também quando a comunicação entre as tarefas acopladas é alta e controladores não são considerados. A heurística desenvolvida se mostra compatível com a utilização de controladores, resultando em *overheads* de comunicação mais baixos que aqueles apresentados pelos algoritmos HEFT e CPOP, utilizados como comparação. Os resultados das simulações realizadas com recursos em topologia de grupos sugerem que o algoritmo é escalável dentro da arquitetura do middleware Xavantes, considerando grupos de recursos heterogêneos conectados por *links* de comunicação heterogêneos. A inserção de uma estratégia de busca de recursos em grupos adjacentes tornou o PCH mais eficiente, superando o desempenho dos algoritmos HEFT e CPOP na grande maioria dos cenários simulados. A complexidade do PCH é $O(rn^3)$, onde r é o número de recursos retornados pelo GM ao escalonador e n o número de nós do grafo, o que o faz um algoritmo aplicável.

Como trabalho futuro podemos enfatizar o desenvolvimento de uma estratégia dinâmica de escalonamento. O algoritmo aqui apresentado é um algoritmo de escalonamento estático, ou seja, o escalonamento de todas as tarefas de um processo é feito antes do início da execução do processo. As principais características das grades computacionais são o seu aspecto dinâmico e a sua heterogeneidade. Então, a adaptação do algoritmo desenvolvido

para promover escalonamento dinâmico, onde tarefas são escalonadas durante a execução do processo, é um importante passo para aumentar a eficiência na execução dos processos.

Outro aspecto a ser desenvolvido engloba o modelo de programação utilizado pelo middleware Xavantes, que permite que o número de tarefas que serão executadas em paralelo em um processo seja determinado em tempo de execução, oferecendo suporte a DAGs dinâmicos e condicionais. O DAG gerado para o escalonamento inicial é baseado em um número estimado fornecido pelo modelo de programação. Se esse número não ocorrer na prática, existirão novas tarefas que não estavam representadas no DAG do processo gerado inicialmente e que devem ser escalonadas em tempo real. Um tratamento mais aprofundado a essa questão pode causar impacto positivo no tempo de execução de processos.

Referências Bibliográficas

[1] Luiz Fernando Bittencourt, Edmundo Roberto Mauro Madeira, Fábio R. L. Cicerre, e Luiz Eduardo Buzato. A path clustering heuristic for scheduling task graphs onto a grid. In *3rd ACM International Workshop on Middleware for Grid Computing, França*, nov., 2005.

[2] Cristina Boeres, José Viterbo Filho, e Vinod E. F. Rebello. A cluster-based strategy for scheduling task on heterogeneous processors. In *16th Symposium on Computer Architecture and High Performance Computing*, pp. 214–221. IEEE Computer Society, out., 2004.

[3] Fábio R. L. Cicerre, Edmundo R. M. Madeira, e Luiz E. Buzato. A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience*, 18(6):581–594, 2006.

[4] Keith Cooper, Anshuman Dasgupta, Ken Kennedy, et al. New grid scheduling and rescheduling methods in the GrADS project. In *18th International Parallel and Distributed Processing Symposium, Estados Unidos*, pp. 199a, IEEE Computer Society, abr., 2004.

[5] Ricardo C. Corrêa, Afonso Ferreira, e Pascal Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, pp. 462, Washington, DC, Estados Unidos, 1996. IEEE Computer Society.

[6] Hesham El-Rewini, Hesham H. Ali, e Ted G. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, 1995.

[7] Carsten Ernemann, Volker Hamscher, e Ramin Yahyapour. Benefits of global grid computing for job scheduling. In *5th International Workshop on Grid Computing, Estados Unidos*, pp. 374–379. IEEE Computer Society, nov., 2004.

[8] Ian Foster. Globus toolkit version 4: Software for service oriented systems. *IFIP International Conference on Network and Parallel Computing*, pp. 2–13, 2005.

[9] Ian Foster e Carl Kesselman. Computational grids. In *Vector and Parallel Processing - VECPAR 2000, 4th International Conference, Porto, Portugal*, pp. 3–37, jun., 2000.

[10] Ian Foster, Carl Kesselman, e Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[11] James Frey. Condor DAGMan: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/, 2002.

[12] Noriyuki Fujimoto e Kenichi Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *2nd International Symposium on Parallel and Distributed Computing, Slovenia*, pp. 80–87. IEEE Computer Society, out., 2003.

[13] Noriyuki Fujimoto e Kenichi Hagihara. A comparison among grid scheduling algorithms for independent coarse-grained tasks. In *Symposium on Applications and the Internet Workshops, Japão*, pp. 674–680. IEEE Computer Society, jan., 2004.

[14] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, e Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, mar., 2004.

[15] Martin Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pp. 280–285, New York, NY, Estados Unidos, 1999. ACM Press.

[16] Tarek Hagras e Jan Janeček. An approach to compile-time task scheduling in heterogeneous computing systems. In *33rd International Conference on Parallel Processing Workshops, Canadá*, pp. 182–189. IEEE Computer Society, ago., 2004.

[17] Mourad Hakem e Franck Butelle. Dynamic critical path scheduling parallel programs onto multiprocessors. In *19th International Parallel and Distributed Processing Symposium, Estados Unidos*. IEEE Computer Society, abr., 2005.

[18] XiaoShan He, XianHe Sun, e Gregor von Laszewski. QoS guided min-min heuristic for grid task scheduling. *Journal of Computer Science and Technology*, 18(4):442–451, 2003.

[19] Edwin S. H. Hou, Nirwan Ansari, e Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[20] Yu-Kwong Kwok e Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[21] Yu-Kwong Kwok e Ishfaq Ahmad. Benchmarking the task graph scheduling algorithms. In *First Merged Symposium. 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP), Estados Unidos*, pp. 531–537, mar., 1998.

[22] Michael J. Litzkow, Miron Livny, e Matt W. Mutka. Condor: A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems, Estados Unidos*, pp. 104–111, jun., 1988.

[23] Cynthia Phillips, Clifford Stein, e Joel Wein. Task scheduling in networks. *SIAM Journal on Discrete Mathematics*, 10(4):573–598, 1997.

[24] Rizos Sakellariou e Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, Estados Unidos*, pp. 111b, IEEE Computer Society, abr., 2004.

[25] Douglas Thain, Todd Tannenbaum, e Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., dez., 2002.

[26] Haluk Topcuoglu, Salim Hariri, e Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[27] B. A. Vianna, Ariel A. Fonseca, N. T. Moura, et al. A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *Proceedings of the 2nd workshop on middleware for grid computing, Toronto, Canadá*, pp. 41–46. ACM Press, out., 2004.

[28] Tao Yang e Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[29] Shaohua Zhang, Yuwei Zong, Zhigang Ding, e Jiamao Liu. Workflow-oriented grid service composition and scheduling. In *International Symposium on Information Technology: Coding and Computing, Estados Unidos*, v. 2, pp. 214–219. IEEE Computer Society, abr., 2005.

Índice Remissivo

activity manager, 18
agrupamento, 35
AM, 18

busca em profundidade, 33, 36

CHP, 12
cluster, 33, 35
clustering, 11, 12, 33
computação distribuída, 1
computação em grade, 5
controladores, 2, 15
  aninhados, 16
  comunicação externa, 40
  comunicação interna, 40
  escalonamento, 30, 33
  paralelos, 15
  seqüenciais, 15
CPOP, 57
custo de computação, 8, 34
custo de comunicação, 8, 34

DAG, 8
dependência de dados, 7, 8
directed acyclic graph, 8

Earliest Start Time - EST, 35
escalabilidade, 7
escalonador, 6
escalonamento, 6
  de controladores, 30, 33, 40
  de tarefas, 7, 11
  dinâmico, 94
  estático, 93
Estimated Finish Time - EFT, 35

GM, 18
grade computacional, 1, 5, 6
  escalonamento de tarefas, 11
  organização virtual, 5
grafo acíclico direcionado, 8
grafos condicionais, 9
grafos dinâmicos, 9
group manager, 18
grupos
  adjacentes, 43
  busca de recursos, 43, 86

HCNFD, 12
HEFT, 12, 56

list scheduling, 11

makespan, 6

nó de entrada, 8
nó de saída, 8

organização virtual, 5

PCH, 33, 42, 93
  agrupamento, 35
  atributos, 34
  complexidade, 46, 53
  definições, 34
  escalonamento de controladores, 40
  seleção de recursos, 37

seleção de tarefas, 35
PM, 18
prioridade, 34
process manager, 18
processo, 7, 8

QoS, 13

sistemas distribuídos, 1
sistemas heterogêneos, 1, 8, 11
escalonamento de tarefas, 11
grade computacional, 1, 5
sistemas homogêneos, 11
clusters, 1
SLR, 56
speedup, 56

tarefas acopladas, 2
task duplication, 11, 12

Weight, 34
workflow, 2, 93

Xavantes, 2, 15, 93
activity manager, 18
escalonamento, 25
group manager, 18
grupos, 18
infra-estrutura, 18
modelo de programação, 8, 15
atividade, 22
processo, 21
process manager, 18
rótulos do DAG, 17, 30