Luis Felipe Sant'Ana

Balanceamento de Carga Dinâmico em Aglomerados de GPUs

Santo André
2014

Luis Felipe Sant'Ana

Balanceamento de Carga Dinâmico em Aglomerados de GPUs

Dissertação Apresentada ao Centro de Matemática, Computação e Cognição da Universidade Federal do ABC, para a obtenção do Título de Mestre em Ciência da Computação.

Orientador: Dr. Raphael Y. de Camargo

Santo André
2014

Sant Ana, Luis F.
Balanceamento de Carga Dinâmico em Aglomerados de GPUs
10 páginas
Dissertação de Mestrado - Centro de Matemática, Computação e Cognição da Universidade Federal do ABC.

1. Programação Paralela
2. Balanceamento de Carga
3. GPGPU

I. Universidade Federal do ABC. Centro de Matemática, Computação e Cognição.

Comissão Julgadora:

Prof. Dr.
XX

Prof. Dr.
YY

Prof. Dr.
Raphael Yokoingawa de Camargo

Resumo

O uso de GPUs em aplicações científicas está cada vez mais difundido, com aplicações em áreas como física, química e bioinformática. Mesmo com o ganho de desempenho obtido com o uso de uma GPU, algumas aplicações ainda requerem elevados tempos de execução. Para esses problemas a utilização de *clusters* de GPUs surgem como uma possível solução. Mas é comum termos máquinas contendo GPUs com variadas capacidades e de diferentes gerações, resultando em *clusters* heterogêneos. Neste cenário, um dos pontos fundamentais é o balanceamento de carga entre as diferentes GPUs, com o objetivo de maximizar a utilização das GPUs e/ou minimizar o tempo de execução da aplicação. Este projeto tem como objetivo o desenvolvimento de um algoritmo de balanceamento de cargas dinâmico entre GPUs heterogêneas. Implementaremos o algoritmo em uma biblioteca existente que facilita o desenvolvimento de aplicações para aglomerados de GPUs e compararemos seu desempenho com outros algoritmos de balanceamento de carga.

Palavras-chave: Programação Paralela, Balanceamento de Carga, GPGPU.

Abstract

The use of GPUs for scientific applications is becoming more widespread, with applications in fields such as physics, chemistry and bioinformatics. Even with the performance gain obtained by using a GPU, some applications require even higher execution times. For these problems the use of GPU clusters arise as a possible solution. But it is common to machines containing GPUs with different capabilities and different generations, resulting in heterogeneous clusters. In this scenario, one of the key points is the load balancing between different GPUs, with the goal of maximizing the use of GPUs and / or minimize the execution time of the application. This project aims to develop an algorithm for dynamic load balancing among heterogeneous GPUs. We implement the algorithm in an existing library that facilitates the development of applications for clusters of GPUs and compare their performance with other load balancing algorithms.

Keywords: Parallel Programming, Load Balancing, CUDA, GPGPU.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Introdução ao balanceamento de carga

A computação paralela vem sendo amplamente utilizada para atender necessidades de aplicações que exigem muito poder computacional. A cada dia surgem propostas de novas máquinas e modelos de arquiteturas paralelas, com diferentes quantidades de processadores, topologias e redes de interconexão. O universo de arquiteturas existentes é grande e elas são normalmente agrupadas como: SIMD e MIMD. Máquinas SIMD possuem um único fluxo de instrução aplicados a múltiplos dados e normalmente são específicas para um tipo de aplicação. As MIMD possuem multiplos fluxos de instrução sendo executados em paralelo. Atualmente as máquinas MIMD são as mais utilizadas e podem ser divididas como: Multi Processadores Simétricos (SMP), Vetoriais, Massivamente paralelas (MPP), agregados de computadores *(Clusters)* e Grades computacionais *(Grids)*. As máquinas SMP, Vetoriais e MPP normalmente fornecem ambientes para programação e uso eficiente, porém são arquiteturas proprietárias e com alto custo de montagem e manutenção. Já os clusters e os grids consistem basicamente de diversas unidades de processamento, que podem ser independentes, interconectadas por uma rede de comunicação. As principais vantagens nesse modelo são custos baixos e alta escalabilidade, isto é, a possibilidade de facilmente alterar, ao longo do tempo, a quantidade de unidades de processamento da arquitetura. No entanto, a computação é baseada em memória distribuída e necessita programação da troca de informações entre os processadores, o que pode ser uma tarefa muito complexa.

As GPUs vêm se mostrando uma excelente alternativa na área de computação de alto-desempenho para aplicações que exigem um alto grau de paralelismo Owens et al. (2008). GPUs modernas podem possuir milhares de núcleos, cada um deles bastante simples, mas que utilizados em paralelo geram um alto poder computacional CUDA Development Core Team (2012).

Muitas aplicações já utilizam o poder de processamento das GPUs, como mecânica dos fluidos Riegel et al. (2009), visualização científica Camargo et al. (2011), aprendizado de máquina Li et al. (2011), bioinformática Borelli et al. (2012) e redes neurais de Camargo (2011). Devido ao relativo baixo custo de placas gráficas contendo GPUs, sua utilização é uma ótima alternativa para pesquisadores pertencentes a instituições com poucos recursos financeiros e com grande necessidade de recursos computacionais.

Em diversos caso, o uso de GPUs ocasionam ganhos superiores a 100 vezes, quando comparado às CPUs tradicionais. No entanto, para muitos problemas, o uso de uma única GPU ainda limita o tamanho e complexidade do problema que pode ser resolvido. Neste caso, é possível utilizar múltiplas GPUs localizados em diferentes computadores, como em *clusters* de GPUs de Camargo (2012); Fan et al. (2004).

Os *clusters* de GPUs são normalmente homogêneos, no qual todas as máquinas apresentam a mesma configuração de hardware. O desenvolvimento de um aplicativo para executar em um *cluster* de GPU é um desafio, porque o programador tem de empregar uma biblioteca de programação paralela, como MPI, além de usar a plataforma CUDA. Existem alguns esforços para oferecer alguns modelos de programação Wang et al. (2011a,b) e bibliotecas Kim et al. (2012); Miyoshi et al. (2012) para o desenvolvimento de aplicações para *clusters* de GPUs. No entanto, a forma mais utilizada para programação para *clusters* de GPUs é combinando CUDA e MPI.

Já no caso de *clusters* heterogêneos é necessário distribuir a carga computacional entre as GPUs. Desenvolver um mecanismo que funcione de modo eficiente para qualquer aplicação é difícil, mas pode-se restringir o mecanismo para alguns tipos específicos de aplicações. Por exemplo, podemos considerar aplicações que possuem um conjunto de dados que podem ser particionados entre as GPUs, utilizando o método de decomposição de domínio. Neste caso, cada GPU fica responsável por uma parte dos dados da aplicação. Muitas aplicações científicas se encaixam neste grupo, incluindo bioinformática Borelli et al. (2012), redes neurais de Camargo (2011), química, física e ciência dos materiais.

Para este tipo de aplicação, a distribuição da carga em *clusters* homogêneos de GPUs é frequentemente obtida com uma divisão simples dos dados de entre as GPUs. No entanto, com grupos heterogêneos de GPUs, esta situação é mais difícil. Existem atualmente várias arquiteturas existentes, como a Tesla, Fermi e Kepler. Essas arquiteturas apresentam diferentes organizações de núcleos e multiprocessadores, quantidades de memória compartilhada por núcleo e velocidades de memória. Devido a essas diferenças, uma divisão da carga com base em simples heurísticas, tal como o número de núcleos na GPU não é eficaz. Na verdade, isso pode levar a um maior tempo de execução de Camargo (2012) comparado a simples ideia de dividir a carga igualmente entre as GPUs heterogêneas. Em um cenário mais geral, uma aplicação pode ter vários *kernels*, cada um com dependências não-lineares entre o tempo de execução e o tama-

nho de entrada.

Apesar da importância do tema, poucos trabalhos apresentam algoritmos de balanceamento de carga em sistemas heterogêneos baseados em GPUs. Os existentes são *frameworks* que requerem a reescrita da aplicação para que seja possível a execução nestes *clusters*.

1.2 Objetivos e Contribuições

O presente trabalho tem como objetivo desenvolver um algoritmo de balanceamento de carga para aglomerados de GPUs heterogêneos em uma biblioteca amplamente utilizada denominada StarPu e comparar com algoritmos de balanceamento de carga atuais.

1.3 Organização da Dissertação

Esta dissertação se divide em seis capítulos. O Capítulo 2 apresenta a fundamentação utilizada neste trabalho. Neste capítulo são discutidos os conceitos, classes e técnicas de escalonamento. O Capítulo 3 descreve os trabalhos relacionados ao que tange o escalonamento de tarefas e mais especificadamente ao balanceamento de carga em aglomerados heterogêneos. O Capítulo 4, mostra os detalhes referentes ao ambiente experimental e implementação. O Capítulo 5 apresenta os resultados e conclusões desta dissertação, assim como sugestões de trabalhos futuros. Por fim o capítulo 6 apresenta o plano de trabalho, das atividades já desenvolvidas e as próximas atividades a serem desenvolvidas.

Capítulo 2

Introdução ao escalonamento

2.1 Escalonamento

Escalonamento é um processo de tomada de decisão que é usado como base em muitas insdústrias manufatureiras e serviços industriais. Ele lida com a alocação de recursos para tarefas a fim de fornecer períodos de tempo a cada tarefa e o fim é otimizar um ou mais objetivos.

Encontrar um escalonamento significa encontrar, para cada tarefa, uma alocação de um ou mais intervalos de tempo, em uma ou mais máquinas. O problema de escalonamento correspondente é encontrar um escalonamento que satisfaça um determinado conjunto de restrições.

Em um ambiente genérico de fabricação, o papel do escalonamento das tarefas é destacado nas ordens de serviço que são lançadas na configuração da fabricação, em forma de tarefas com datas de entrega associadas. Essas tarefas frequentemente devem ser processadas em máquinas em uma dada ordem ou sequência. Os processamentos das tarefas podem atrasar, se certas máquinas estiverem ocupadas. Eventos imprevistos no chão-de-fábrica, tais como quebra de máquinas ou tempos de processamento maiores que os previstos, também devem ser levados em consideração, desde que esses eventos venham a impactar diretamente o escalonamento das tarefas. Neste ambiente, o desenvolvimento de um escalonador de tarefas detalhado ajuda a manter a eficiência e o controle das operações.

O chão-de-fábrica não é a única parte da organização que impacta o processo de escalonamento. O escalonador também é afetado pelo processo de planejamento da produção que lida com o planejamento a médio e a longo prazos para toda a organização. Esse processo tenta otimizar toda linha de produtos da empresa e a alocação de recursos baseados em seus níveis de estoque, previsões de demanda e necessidades de recursos. As decisões tomadas neste nível mais alto de planejamento podem impactar o processo de escalonamento diretamente.

Um exemplo de escalonamento é na indústria de semicondutores no qual o objetivo é maximizar o tempo de utilização dos equipamentos e minimizar o tempo ocioso e de configuração dos mesmos.

2.1.1 Notação dos problemas de escalonamento

Associados às tarefas, existem as seguintes informações:

- Tempo de execução (p_{ij}) tempo necessário para a execução da tarefa t_i na máquina m_j. Denotado apenas por p_i se todas as máquinas forem idênticas
- Data de disponibilidade (r_i) instante em que a tarefa t_i se torna disponível para ser executada
- Prazo (d_i) instante de tempo no qual a tarefa t_i deve estar pronta
- Peso (w_i) normalmente indica um fator de prioridade da tarefa t_i

A notação utilizada para representar os problemas de escalonamento, é representada pela tripla:
$\alpha|\beta|\gamma$
α que descreve os recursos disponíveis
β que descreve as tarefas a serem executadas
γ que descreve o critério de otimização
α pode ser representado por:

- 1 uma única máquina
- P ou P_m m máquinas paralelas idênticas cada tarefa t_i pode ser processada em qualquer uma das m máquinas por p_i unidades de tempo, se uma tarefa só puder ser executada em um subconjunto das máquinas, a restrição será indicada no campo
- $P\infty$ ou $\overline{P}$ é o número ilimitado de máquinas idênticas

β pode ser representado por:

- Q_m máquinas paralelas uniformes no qual m máquinas operam com velocidades iguais (a máquina m_j opera com velocidade v_j) $p_{ij} = p_i = v_j$
- R_m máquinas paralelas não-relacionadas, m máquinas diferentes em paralelo
- $p_{ij} = p_i = v_{ij}$, onde v_{ij} é a velocidade da tarefa t_i na máquina m_j

Em sistemas de produção as máquinas são consideradas dedicadas, não paralelas. Além disso, uma tarefa t_i é composta por um conjunto de operações denotadas por $O_{i1}|O_{i2}|...|O_{in}$.

F_m representa o modelo flow-shop, no qual todas as tarefas possuem o mesmo número de operações, que devem ser executadas na mesma ordem por todas as máquinas, em série.

O_m representa o modelo open-shop, como um flow-shop, mas não há ordem de precedência entre as operações

J_m representa o modelo job-shop. Neste modelo cada tarefa possui um roteiro pré-determinado se uma tarefa puder executar em uma mesma máquina mais de uma vez, dizemos que existe recirculação (denotado por $recrc$)

r_i é a data de disponibilidade. Se r_i não estiver no campo, tarefas podem ser iniciadas a qualquer momento. Se r_i estiver no campo, então a tarefa t_i não pode ser iniciada antes do tempo r_i

$pmtn$ representa interrupção. O processamento de uma tarefa pode ser interrompido e retomado do ponto onde parou, em qualquer máquina.

s_{ik} é o tempo de preparação da máquina (setup). Uma máquina que executou uma tarefa t_i precisa de s_{ik} unidades de tempo para ser preparada antes de executar uma outra tarefa t_k

$prec$ relações de precedência. Uma relação de precedênca entre duas tarefas indica que uma tarefa não pode ser iniciada antes do término da execução da outra. Representadas por um grafo de precedências, onde um vértice representa uma tarefa e um arco entre t_i e t_k indica que t_k só pode ser executada após o termino de t_i.

c_{ik} tempo de comunicação entre máquinas. A tarefa t_k depende do resultado da execução de t_i . Se t_k não for escalonada na mesma máquina que t_i, então deverá esperar mais c_{ik} unidades de tempo antes de poder ser executada.

2.1.2 Classes de Escalonamento

Escalonamento sem atrasos

Definição: Um escalonamento válido é dito sem atraso, se nenhuma máquina fica inativa, quando existem tarefas disponíveis para serem executadas.

Escalonamento ativo

Definição: Um escalonamento realizável e não-preemptivo é dito ativo se não for possível, apenas trocando a ordem das tarefas/operações em uma máquina, construir um outro escalonamento onde ao menos uma tarefa termine mais cedo e nenhuma outra tarefa seja atrasada.

Escalonamento semi-ativo

Definição: Um escalonamento realizado e não-preemptivo é dito semi-ativo se nenhuma tarefa/operação pode terminar mais cedo sem que a ordem do processamento das tarefas de alguma máquina seja mudada.

2.1.3 Classificação dos problemas de escalonamento

Os problemas de programação de operações em máquinas vêm sendo caracterizados por diversos autores em diferentes formas, dentre eles Baker, 1974; Blazewicz et al., 1996; Conway et al., 1967; French, 1982; Graves, 1981 e Pinedo, 2008.

Em situações de escalonar tarefas nas máquinas disponíveis surgem problemas complexos. Pois, as restrições tecnológicas e a medida de desempenho do escalonador devem ser especificadas. As restriçõs tecnológicas são determinadas principalmente pelo fluxo das tarefas nas máquinas.

Neste contexto, Maccarthy e Liu (1993) classificam os problemas de programação de operações da seguinte forma:

- **Máquina única** - existe somente uma única máquina disponível para a execução das tarefas;
- **Flow shop** - em que todas as tarefas possuem o mesmo fluxo de processamento em todas as máquinas;
- **Job shop** - em que todas as tarefas possuem um roteiro específico de processamento, determinado para cada tarefa;
- **Open shop** - em que não existem roteiros de processamento preestabelecidos para as tarefas;
- **Flow shop permutacional** - flow shop onde a ordem de processamento das tarefas é exatamente a mesma para todas as máquinas;

O problema do sequenciamento em uma única máquina é frequentemente muito simples e quase sempre parte de um problema de programação complexo. Segundo Pinedo (2008), os problemas do sequenciamento em uma única máquina muitas vezes têm propriedades que os de em máquinas em paralelo ou em série não possuem. Os resultados que podem ser obtidos para os problemas do sequenciamento em uma única máquina não só fornecem o conhecimento para o ambiente de uma única máquina, como também fornecem base para heurísticas aplicáveis a ambientes mais complexos.

Na prática, os problemas de escalonamento em ambientes mais complicados são frequentemente decompostos em subproblemas de uma única máquina.

Por exemplo, um ambiente complexo, com um único gargalo, pode dar origem a um modelo de sequenciamento em uma única máquina. Dessa forma, o problema do sequenciamento em uma única máquina é importante por diversas razões, dentre elas pode-se citar:

- O processo de aprendizado, já que o problema de escalonamento em uma única máquina pode ilustrar uma variedade de tópicos de escalonamento tornando modelos tratáveis. Esse problema fornece um contexto para que se investigue muitas medidas de desempenho e técnicas de solução. Além disso, é uma base para o entendimento de conceitos de escalonamento úteis para modelar sistemas mais complexos.

- Para entender completamente o comportamento de um sistema complexo, é vital entender como funciona cada um de seus componentes e muito frequentemente o problema de uma única máquina aparece como componente elementar em um problema de escalonamento maior.

- Algumas vezes é possível resolver o problema de escalonamento em uma única máquina independentemente e então incorporar o resultado em um problema maior. Por exemplo, em um processo com múltiplas operações, frequentemente existe uma operação gargalo e o tratamento dessa operação gargalo, vista como uma análise de um problema de uma única máquina, determina as propriedades de todo o escalonamento.

2.1.4 Definição de atributos

Ao lidar com os atributos para o modelo de uma única máquina, é útil distinguir entre informações conhecidas previamente e informações que são geradas como resultados de decisões de escalonamento. A informação que é conhecida previamente serve como parâmetro de entrada para função de escalonamento e é usualmente conveniente usar letras minúsculas para denotar esse tipo de informação. As três informações básicas que ajudam a descrever trabalhos no problema básico determinístico de uma única máquina são:

- Tempo de processamento (t_j): tempo de processamento requerido pelo trabalho j;

- Data inicial (r_j): o ponto no tempo em que a tarefa j está disponível para processamento; e

- Prazo (d_j): é o ponto limite no tempo em que o processamento da tarefa j precisa ser concluída.

Os prazos podem não ser pertinentes em certos problemas, mas estabelecer os prazos (deadlines) é um problema comum na indústria e o problema básico pode auxiliar na determinação do prazo de entrega. É conveniente usar letras maiúsculas para denotar as informações resultantes do escalonamento.

Tempo de Conclusão (C_j). O tempo no qual o processamento do trabalho j é terminado.

Os critérios quantitativos para escolher uma sequência são geralmente funções dos tempos de conclusão. Duas funções importantes são:

Tempo de Fluxo (Flowtime) – F_j – Tempo total que o trabalho j fica no sistema:

$F_j = C_j – r_j$

Defasagem (Lateness) – L_j – Diferença entre a data de conclusão e o prazo do trabalho j, podendo assumir valores positivos ou negativos

Esses dois valores refletem dois critérios importantes. O tempo de fluxo que mede a resposta do sistema e representa o tempo que uma tarefa leva entre sua chegada e sua saída. A defasagem, L_j, que mede a conformidade do escalonamento em relação ao prazo. É importante notar que a defasagem terá valor negativo quando uma tarefa é finalizada antecipadamente. Defasagens negativas podem representar serviços melhores do que solicitados, enquanto atrasos positivos representam, quase sempre, serviços piores do que requisitados. Em muitas situações, penalidades distintas e outros custos serão associados para defasagens positivas e para defasagens negativas. Dessa forma, tem-se as definições de Atraso (Tardiness) e Antecipação (Earliness):

Atraso (Tardiness) – T_j – é o quanto o trabalho j atrasou em relação ao seu prazo, caso contrário será considerado zero:

$T_j = max\{0, L_j\}$

Antecipação (Earliness) – E_j – é o quanto o trabalho j é antecipado em relação ao prazo, caso contrário será considerado zero:

$E_j = max\{d_j – C_j, 0\}$

Makespan – C_{max} – é definido como o maior dentre as datas de conclusão ($C_1, ..., C_n$), ou seja, ao tempo de conclusão do último trabalho a sair do sistema ($C[n]$). Minimizar o makespan usualmente implica em uma boa utilização dos recursos.

The Total Weighted Completion Time (Tempo total de conclusão ponderado)
$1||sum(w_jC_j)$

Primeiro tempo de processamento com menor custo. (WSPT) Teorema: WSPT é ótima para $1||sum(w_jC_j)$. Neste teorema supomos que temos várias cadeias, e não pode parar de processar uma cadeia que já começou. Nos ajuda decidir qual Cadeia de tarefas precisamos processar primeiro para minimizar o tempo total de processamento.

Supondo agora que o escalonador não precisa terminar todas as tarefas de uma cadeia para começar a processar a outra cadeia. Basicamente este lema nos diz que não faz sentido já que iniciamos a processar uma cadeia interromper para processar outra cadeia.

Algorithm 3.1.4 (Total Weighted Completion Time and Chains)

Sempre que a máquina é liberado, selecione entre as cadeias restantes aquele com o maior ρ-factor. Processar esta cadeia, sem interrupção até e inclusive o trabalho que determina o seu ρ-factor.

The Number of Tardy Jobs

As tarefas tem datas limites para terminar. A ideia do algoritmo é ordenar as tarefas baseado na data limite de cada tarefa. Deleta-se a tarefa mais longa, caso alguma tarefa tenha a data limite ultrapassada. Se colocarmos pesos, é equivalente ao problema da mochila.

The Total Tardiness - Dynamic Programming

Minimizar o número de trabalhos atrasados sum(Tj), NP-Dificil, usa-se a programação dinâmica para resolver (algoritmo pseudo-polinomial).

The Total Tardiness - An Approximation Scheme

Se fez um algoritmo de tempo polinomial que se aproxima da solução ótima. Usa o Esquema de aproximação em tempo polinomial completo.

The Total Weighted Tardiness (problema fortemente NP-Difícil)

É o mesmo problema anterior agora com ponderação, problema levanta bastante atenção entre os pesquisadores. O problema da 3-Partição (Dividir um conjunto de inteiros em três partes, no qual a soma de cada parte seja a mesma) se reduz a este problema.

Resumo

$1||Somaw_jC_j$ Regra WSPT, funciona para chains tambem $1||Somawj(1-e-rC_j)$ Regra WDSP, para cadeias também $1||L_{max}$ Regra EDD (Primeiro tarefa que termina primeiro) $1|prec|h_{max}$ Algorithm 3.2.1. $1|rj|L_{max}$ Branch and bound $1||SU_j$ Algorithm 3.3.1. $1||ST_j$ Dynamic programming Alg. 3.4.4.

The Total Earliness and Tardiness

Soma total earliness e total tardiness. Depois que uma tarefa se inicia não se pode mais parar o processamento, gera-se o escalonamento ótimo.

Primary and Secondary Objectives

Mais realista, pois na pratica temos mais de um objetivo. A ideia é encontrar a melhor solução para um objetivo e a partir dessa solução encontrar uma boa solução na segunda função objetivo. Temos neste caso dois objetivos a serem minimizados. São NP-Hard.

Multiple Objectives: A Parametric Analysis

Escalonamento pareto-ótimo – não é possível diminuir o valor de uma função objetivo sem diminuir o valor da outra.

Job Families with Setup Times

Tarefas que fazem parte da mesma família podem ser processadas sem a adição de tempo de configuração (podem apresentar tempo de processamento diferentes). Mas se há a troca de uma família f para uma h, um tempo s de configuração é necessário. No exemplo com pesos no tempo total de processamento a resolução se deu utilizando programação dinâmica.

Batch Processing

Máquina processa um número de tarefas simultaneamente (um batch de tarefas ao mesmo tempo). Comum na industria.

Modelos de máquina paralela, primeiro determinamos/alocamos os recursos em cada máquina, depois o escalonamento em cada maquina.

A preempção tem um papel mais importante que nos modelos de única máquina, mesmo quando as tarefas são realizadas no mesmo tempo.

Escalonamento Offline – todos os dados (tempo processamento, prazos, periodo de submissão da tarefa) são conhecidos em detalhes e podem ser considerados no processo de otimização.

Escalonamento On-line – Os dados não são conhecidos a priori, o tempo de processamento é conhecido apenas depois que a tarefa termina, e a hora que a tarefa foi submetida apenas quando a tarefa é efetivamente submetida.

2.1.5 Exemplos de modelos de escalonamento

The Makespan without Preemptions
($P_m||C_{max}$)

Utiliza o tempo máximo para completar a tarefa. P_m indica máquinas em paralelo. Problema interessante, pois tem o efeito de balanceamento de carga sobre várias maquinas, importante objetivo na pratica.

É equivalente ao problema de partição, é NP-Dificil. É NP-Dificil devido as diferentes combinações de escalonamento mas maquinas. Problema resolvido através de heurística. A huerística chama-se Primeiro Tempo de Processamento Mais Longo (LPT).

CP rule – O nível mais alto primeiro para tempo de processamento igual a 1 e com precedência, é ótimo. LNS é equivalente a CP Rule, não é necessariamente ótima com restrições de precedência arbitrária.

LFJ - Least Flexible Job first . Para o caso de restrições de subconjunto de maquinas a serem processadas, seleciona as maquinas disponíveis no menor numero de maquinas.

The Makespan with Preemptions

Mesmo exemplo, mas agora com preempção permitida $P_m|prmp|C_{max}$. Tratado como um problema de programação linear.

The Total Completion Time without Preemptions

$P_m||Sum(C_j)$ – SPT – A tarefa mais curta primeiro tem escalonamento ótimo. $P_m|prec|Sum(C_j)$ – agora temos precedência na ordem das tarefas, é NP-Dificil, restrições de precedência arbitrária

The Total Completion Time with Preemptions

$Q_m|prepr|Sum(C_j)$ – Qm indica maquinas com diferentes velocidades de processamento. Shortest Remaining Processing Time on the Fastest Machine = tarefa mais curta para a maquina mais rapida e assim por diante.

Online Scheduling Escalonamento online, ou seja, ao mesmo tempo que estamos escalonando.

Máquinas em série, ideia da manufatura, que existe uma sequencia de máquinas que a matéria-prima precisa passar para se tornar um produto final. Flexible Shop são

máquinas em série com vários estágios, cada estágio tem várias máquinas em paralelo. Objetivo para esse ambiente é o Makespan (tempo total C_{max}).

O escalonamento flow shop é um sistema de trabalho de tarefas em máquinas em série, onde cada tarefa tem que ser processada em cada uma das máquinas. Todas as tarefas devem seguir a mesma rota, ou seja, elas têm que ser processadas, primeiro na máquina 1, depois na máquina 2, e assim por diante. Após a conclusão de uma tarefa em uma máquina, a tarefa se junta à fila da próxima máquina. O problema de escalonamento flow shop é classificado como NP-difícil para a maioria dos problemas clássicos

O Job-shop Scheduling Problem (JSP) pode ser descrito por: n tarefas, onde cada tarefa é composta por j operações que devem ser processadas em m máquinas. Cada operação j utiliza uma das m máquinas com um tempo de processamento fixo. Cada máquina processa uma operação por vez sendo que não ocorre interrupção. As operações devem ser processadas em ordem através de um roteiro pré-estabelecido. O problema consiste em encontrar um escalonamento que obedeça as precedências de operações nas máquinas tal que minimize o makespan C_{max} dado o processamento da ultima tarefa.

Job Shop é um Flow Shop, no qual as tarefas tem rotas diferenciadas, mas pré-determinadas.

No Open-Shop, a rota das tarefas são abertas, não é pré-determinada. Open Shop: O ambiente de máquinas típico de open shop caracteriza-se pelo fato de não existir uma sequência pré-definida a ser seguida por cada tarefa j. Posto de outra forma, é fornecido ao escalonador liberdade de escolha da sequência a ser seguida por cada tarefa j ao longo do sistema.

2.2 GPUs

As GPUs vêm se mostrando uma excelente alternativa na área de computação de alto-desempenho para aplicações que exigem um alto grau de paralelismo Owens et al. (2008). GPUs modernas possuem dezenas de núcleos, cada um deles bastante simples, mas que utilizados em paralelo geram um alto poder computacional CUDA Development Core Team (2012).

Muitas aplicações já utilizam o poder de processamento das GPUs, exemplos são mecânica dos fluidos Riegel et al. (2009), visualização científica Camargo et al. (2011), aprendizado de máquina Li et al. (2011), entre outras. Devido ao relativo baixo custo de placas gráficas contendo GPUs, sua utilização é uma ótima alternativa para pesquisadores pertencentes a instituições com poucos recursos financeiros e com grande necessidade de recursos computacionais.

2.3 Suporte ao Paralelismo de Tarefas

Diversas ferramentas de programação suportam paralelismo de tarefas em CPUs multicore. Recentemente, alguns trabalhos em andamento têm buscado oferecer suporte a esse paradigma também em sistemas híbridos compostos por CPUs e GPUs.

2.3.1 StarPU

O StarPU é uma ferramenta para programação paralela que oferece suporte para arquiteturas híbridas, como CPUs multicore e aceleradores. O StarPU propõe uma abordagem de tarefas independente da arquitetura base. São definidos codelets como uma abstração de uma tarefa que pode ser executada em um núcleo de uma CPU multicore ou submetido a um acelerador. Cada codelet pode ter múltiplas implementações, uma para cada arquitetura em que o codelet pode ser executado, utilizando as linguagens ou bibliotecas específicas para a arquitetura alvo. Uma aplicação StarPU é descrita como um conjunto de codelets com suas dependências de dados (AUGONNET; NAMYST, 2009).

A ferramenta possui um conjunto de políticas de escalonamento implementadas que o programador pode escolher de acordo com as características da aplicação. A principal delas faz uso do algoritmo de escalonamento estático HEFT (Heterogeneous Earliest Finish Time) para escalonar as tarefas com base em modelos de custo de execução das tarefas.

2.3.2 Charm++

No Charm++ as tarefas são representadas pelos chares, que são objetos paralelos e representam unidades locais de trabalho. Cada chare possui dados locais, métodos para tratamento de mensagens e possibilidade de criar novos chares, assim como processos MPI-2. Existe ainda um tipo especial de chare, chamado branch-office, que possui uma ramificação em cada processador e um único nome global. Branch- office chares oferecem métodos sequenciais que podem ser acessados por chares de forma transparente em qualquer processador.

No Charm++ a sincronização pode ser feita através de Futures, objetos de comunicação, replicados ou compartilhados. O future é uma estrutura que serve para armazenar um valor que pode ser acessado no futuro por outro chare. O acesso a essa estrutura é bloqueante e pode ser comparado a um acesso ao resultado de um método spawned precedido de uma chamada sync. A utilização de objetos de comunicação permite que um chare se comunique por troca de mensagens, tornando a comunicação semelhante à realizada com MPI. Os demais objetos introduzem conceitos para comunicação não encontrados nos outros ambientes.

O Charm (KALE et al., 1995) foi uma das primeiras implementaçõses do conceito de Atores, que são objetos concorrentes que se comunicam apenas por troca de mensagens. Charm++ (KALE; KRISHNAN, 1996) é baseado no Charm e suporta diferentes modos de compartilhamento de informações. Ele reúne recursos propostos em outros ambientes, como objetos sequenciais e paralelos, comunicação por troca de mensagens e futures. O Cilk (BLUMOFE et al., 1995) é um ambiente de programacção paralelo focado em arquiteturas SMP. Ele é baseado na linguagem C. O Cilk oferece também um escalonador de processos que é provado ser eficiente (VEE; HSU, 1999). Baseando-se no Cilk, foi desenvolvido um ambiente para execução de programas em arquiteturas com memória distribuída e aproveitamento de recursos ociosos, chamado Cilk-NOW (BLUMOFE; LISIECKI, 1997).

2.3.3 Kaapi

O XKaapi (INRIA; MOAIS; LIG, 2011) é uma reimplementação do KAAPI com suporte a paralelismo de tarefas ”de grão fino”. O KAAPI (GAUTIER; BESSERON; PIGEON, 2007) (Kernel for Adaptative, Asynchronous Parallel and Interactive programming) é uma ferramenta para computação paralela em CPUs multicore e clusters. A implementação atual do XKaapi oferece suporte a arquiteturas multicore e extensões para suporte eficiente a GPUs foram propostas em (HERMANN et al., 2010; LIMA et al., 2012). O XKaapi é composto por um conjunto de APIs (Application Programming Interfaces) e pelo kernel, um ambiente de execução para as APIs que oferece escalonamento baseado em roubo de tarefas. A Kaapi++ é a interface do XKaapi baseada em um DFG (Data Flow Graph) para C++ e é dividida em três componentes: a assinatura da tarefa (task signature) onde são definidos o número e as características dos parâmetros da tarefa; a implementação da tarefa (task implementation) que especifica a implementação da tarefa para uma arquitetura e a criação da tarefa (task creation) que submete a tarefa para a pilha de execução.

2.3.4 Cilk

A ferramenta para programação paralela Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998; GROUP, 2012) é uma extensão da linguagem C para permitir a submissão, execução e sincronização de tarefas paralelas. A implementação de Cilk é baseada no algoritmo de escalonamento dinâmico por meio de roubo de tarefas apresentado em (BLUMOFE; LEISERSON, 1994). Cilk define tarefas como funções individuais que podem submeter novas tarefas dinamicamente. A sincronização é feita permitindo que as tarefas esperem pelas tarefas filhas, ou seja, tarefas que foram submetidas pela tarefa original.

2.3.5 Intel TBB

O Intel TBB (Threading Building Blocks) (REINDERS, 2010; CORPORATION, 2012) é uma biblioteca baseada em templates para programação paralela em C++ que faz uso de threads. Essa biblioteca permite expressar o paralelismo em diversos paradigmas de programação paralela como o paralelismo de dados, de laços ou de tarefas. As unidades de trabalho paralelas resultantes são escalonadas em threads por meio de um algoritmo de roubo de tarefas inspirado no ambiente Cilk.

2.3.6 OpenMP

OpenMP (Open Multi-Processing) (CHAPMAN; JOST; PAS, 2007) é uma ferramenta para programação paralela baseada na adição de diretivas de compilação em códigos C, C++ e Fortran. As diretivas OpenMP são utilizadas para indicar a paralelização de laços ou trechos de código. A partir da versão 3.0 (OpenMP ARB, 2008) foi inserido o conceito de tarefas, o que permite a utilização do paralelismo de tarefas através do uso de diretivas para delimitação de trechos de código como unidades de trabalho paralelas (AYGUADE et al., 2009). A especificação OpenMP (OpenMP ARB, 2008) não define uma política para o escalonamento das tarefas, ficando essa escolha a cargo de cada implementação.

Capítulo 3

Trabalhos Relacionados

3.1 Introdução

Esta seção apresenta os tabalhos relacionados ao presente trabalho.

O problema do balanceamento de carga é estudado por pesquisadores da área de Teoria de Escalonamento há mais de 50 anos. Já em 1966, Graham (1966) estudou o que chamou de ”anomalias” na execução de tarefas em processadores de velocidades diferentes. Ele mostrou, por exemplo, que a adição de novos processadores ou a troca de alguns processadores por outros de maior velocidade podem propiciar o desbalanceamento de carga e, portanto, não necessariamente implicam em um ganho no desempenho. Logo ficou clara a necessidade de algoritmos mais sofisticados para lidar com esse problema. Por favor, consulte Leung (2004) para uma introdução sobre Teoria do Escalonamento.

As novas possibilidades trazidas com as novas arquiteturas de GPU estão instigando pesquisadores a rever o problema do balanceamento de carga neste novo contexto. Apesar do balanceamento de carga ser crítico para o desempenho de aplicações paralelas, como no caso de aplicações gráficas Foley and Sugerman (2005); M. Mller et al. (2007), ainda não há na literatura muitos estudos sobre balanceamento de carga em GPUs.

Cederman and Tsigas (2008) analisaram várias estratégias de balanceamento de carga estática e dinâmica para um problema de partição de octree em GPUs. O balanceamento de carga é realizado internamente em uma única GPU, sem interação com a CPU. Em outro estudo Guevara et al. (2009), os autores propõem a obtenção das cargas de trabalho de tarefas em múltiplos núcleos e sua fusão em um *super-kernel*. No entanto, esta fusão precisa ser realizada estaticamente, e assim o equilíbrio dinâmico de carga não pode ser sempre garantido.

O problema de distribuição de carga em GPUs heterogêneas começou a ser estudado recentemente. Acosta et al. (2012) propôs um algoritmo de balanceamento

de carga dinâmico, que busca, interativamente, uma boa distribuição de trabalho entre as GPUs durante a execução da aplicação. Os autores avaliaram os problemas de multiplicação de matrizes e alocação de recursos. O método iterativo pode demorar para atingir uma distribuição de carga adequada, reduzindo o desempenho da aplicação. Além disso, não é possível utilizar informações relativas a tempos de execução anteriores para gerar um perfil de execução para as futuras execuções da apliação.

Em outro trabalho, de Camargo (2012) propõe um algoritmo que determina a distribuição, antes do início da execução da aplicação, usando perfis estáticos obtidos em execuções anteriores. O algoritmo foi avaliado para uma aplicação de redes neurais de grande escala. O algoritmo encontra a distribuição de dados que minimiza o tempo de execução da aplicação, garantindo que todas as GPUs gastem a mesma quantia de tempo realizando o processamento de *kernels*. Esta abordagem, entretanto, não permite alterações dinâmicas na distribuição dos dados.

Algoritmos de distribuição de carga são frequentemente baseados no conceito de tarefas. Neste caso, as tarefas das aplicações podem ser dinamicamente distribuídas entre os SMs (*Streaming Multiprocessor*) das GPUs Chen et al. (2010), ou entre diferentes GPUs Augonnet et al. (2010) para realizar a distribuição da carga. Neste contexto surge o modelo de programação de uso geral para sistemas multi-core heterogêneos chamado *Merge* Linderman et al. (2008). O objetivo do *framework* é substituir as atuais abordagens *ad hoc* para programação paralela em plataformas heterogêneas, com uma metodologia baseada em uma biblioteca que pode distribuir automaticamente o cálculo através de núcleos heterogêneos.

StarPU Augonnet et al. (2010) é um *framework* para o escalonamento de tarefas em plataformas heterogêneas, onde informações sobre o modelo de desempenho das tarefas podem ser passadas para orientar as políticas de escalonamento. Os dois princípios básicos da StarPU são (i) as tarefas podem ter várias implementações, para alguns ou para cada uma das várias unidades de processamento heterogêneas disponíveis na máquina, e (ii) a transferência de dados para estas unidades de processamento são tratadas de forma transparente pelo StarPU. Escalona dinamicamente tarefas em todos as unidades de processamento. Evita as transferências de dados desnecessário entre aceleradores. Aceite tarefas que podem ter várias implementações. Fornece uma camada alto nível de gerenciamento de dados. Quando uma tarefa é submetida, pela primeira vez entra em um ”pool”de ”Tarefas congelados”até que todas as dependências sejam atendidas. Em seguida, a tarefa é ”empurrado”para o escalonador. Algumas politicas de escalonamento foram testadas utilizando a interface Starpu, entre elas greedy, no-prio, ws, w-rand, heft-tm.

A estratégia greedy, no-prior e ws são estratégias gulosas, no qual a greedy se diferencia da no-pior pois a greedy tem suporte a prioridades, então pode ponderar pesos de prioridade a tarefas enquanto na no-prior não é possível. E a ws é uma

estratégia gulosa baseada no ”work-stealing”que é uma estratégia que consiste em as unidades de processamento terem uma pilha de tarefas e se houver o término das tarefas pode haver o ”roubo”da tarefa de outras unidades.

Na estratégia w-rand, cada unidade de processamento está associada com uma relação que pode ser considerada como um fator de aceleração. Cada vez que uma tarefa é submetida, uma das unidades de processamento é selecionada com probabilidade proporcional à sua relação. Essa relação pode, por exemplo, ser definida pelo programador, ou ser medida com benchmarks de referência. A estratégia w-rand é normalmente adequado para tarefas independentes de igual tamanho.

A tarefa de distribuir uniformemente sobre as unidades de processamento baseado na velocidade não significa necessariamente fazer sentido para as tarefas que não são igualmente custosas, ou quando existem dependências entre eles. Assim, os programadores podem especificar um modelo de custo para cada tarefa. Este modelo pode por exemplo representar a quantidade de trabalho e ser usado em conjunto com a velocidade relativa de cada um das unidades de processamento. É também possível modelar diretamente o tempo de execução de cada uma das arquitecturas. Usando esses modelos, implementa-se a estratégia heft-tm. Dada a sua duração esperada nas várias arquiteturas, é atribuída uma tarefa para as unidades de processamento que minimiza o tempo de término, no que diz respeito à quantidade de trabalho já atribuída a esta unidade de processamento.

Charme++ é uma biblioteca em C++ que fornece sofisticados mecanismos de balanceamento de carga e otimização de mecanismos de comunicação. Sua implementação do sistema em tempo de execução suporta GPUs e processadores ”Cell”.

Diamos and Yalamanchili (2008) apresentar um conjunto de técnicas para o sistema de execução ”Harmony”que pretende implementar em um sistema completo. Algumas dessas técnicas são semelhantes às aplicadas no StarPU . Por exemplo , eles consideram o problema de escalonamento de tarefas com o apoio de modelos de desempenho . No entanto, ”Harmony”não aceita as estratégias de programação fornecidas pelo usuário . Além do modelo baseado em regressão também proposto por Harmony, a forte integração da biblioteca de gerenciamento de dados do StarPU juntamente com o programador é mais simples e mais preciso que modelos de desempenho baseadas no histórico.

StarSs Ayguade et al. (2009) introduz as anotações ”#pragma”. Eles contam com um compilador ”source-to-source”para gerar tarefas. Contrariamente ao StarPU , a implementação do modelo StarSs é feito por meio de um sistema de execução separado para cada plataforma. Alguns esforços são feitos para combinar esses diferentes sistemas de tempo de execução: Planas et al. permite que os programadores incluiam tarefas dentro tarefas SMPSs, mas este continua a ser responsabilidade do programador decidir quais tarefas devem ser inseridas. Em contraste, StarPU considera as tarefas que podem ser executadas com indiferença em múltiplos alvos, leva em consideração

a heterogeneidade do sistema.

A abordagem do sistema de execução Anthill é muito semelhante ao StarPU . Teodoro et al. (2009) experimentaram o impacto da programação de tarefas para clusters de máquinas equipadas com processadores com uma única GPU e com múltiplos processadores. Eles implementaram duas políticas de escalonamento que são equivalentes às estratégias gulosas simples. Anthill considera apenas speedups relativos para selecionar a unidade de processamento mais adequada.

Capítulo 4

Algoritmo Proposto

4.1 Introdução

Esta seção demonstra detalhes de implementação do algoritmo proposto para o balanceamento de carga dinâmico em aglomerados de GPUs.

Em uma típica aplicação paralela, os dados da aplicação são divididos entre as threads em um processo chamado decomposição de domínio. As threds entao simultenamente processam parte dos dados. Depois de terminarem, mesclam os resultados processados e terminam a aplicação ou continuam para a próxima fase de computação. A tarefa do algoritmo proposto é determinar o tamanho do bloco de dados atribuído a cada GPU e CPU no sistema. O termo processador significará um única CPU ou GPU.

O algoritmo tem três partes, que são: (1) modelo de desempenho do processador, onde um modelo de execução é determinado durante a execução da aplicação; (2) seleção do tamanho ótimo do bloco, onde baseado no modelo de performance o algoritmo seleciona a melhor distribuição de tamanho de blocos entre os processadores; e (3) o rebalancemaentom onde o algoritmo recalcula o tamanho do bloco ótimo quando a diferença no tempo de execução nos diferentes processadores é maior que um limiar.

Modelo de desempenho do processador. Inicialmente determina-se a função $P_p[x]$, que fornece a velocidade de processamento para um dado bloco de tamanho x no processador p. Para construir estas curvas, um bloco de tamanho $initialBlockSize$, definido pelo usuário é alocado para cada processador. Seleciona-se o processador com o menor tempo de processamento t_f e dobra-se o tamnho do bloco para este processador. Para os outros processadores, com o tempo de término t_i, selciona-se o tamanho do bloco proporcional a razão t_i/t_f.

A figura 4.1 mostra um exemplo de media de tempo para uma GPU e uma CPU para diferentes tamanhos de bloco. Pode-se notar que as curvas podem ser aproximadas por funções polinimiais. Nestas curvas encontrou-se as melhores curva que se ajustam através do método dos mínimos quadrados, usando uma função da forma

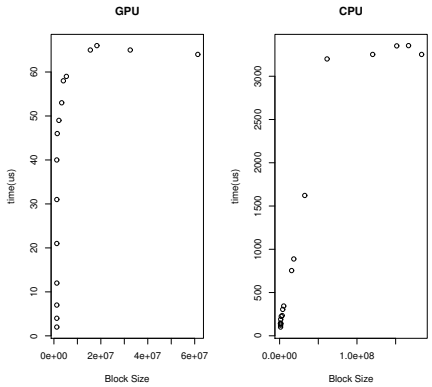

Figura 4.1: Curvas para GPU x CPU

$y = a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + ... + a_n f_n(x)$. A curva é ajustada após a geração de poucos pontos, por exemplo quatro, gerando um modelo de tempo de execução para cada processador.

A figura 4.1 mostra um exemplo d medida de tempo de processamento para uma GPU e uma CPU. pra diferentes tamnhos de bloco em dias aplicações. Pode-se notar que as curvas para aplicação blackscholes pode ser aproximada por funções lineares, enquanto para a multiplicação de matrizes usa-se uma função exponencial. Encontrou-se o melhor ajuste das curvas utilizando o método dos mínimos quadrados:

$$y = a_1 f_1(x) + a_2 f_2(x) + ... + a_n f_n(x) \tag{4.1}$$

onde $f_i(x)$ corresponde a um pequeno conjunto de funções de complexidade comum, tais como x, x^2, e^x, lnx, etc.

Algorithm 1 Modelo de desempenho do processador

function determineModel()
$blockSizeList \leftarrow initialBlockSize$;
while $fitValues.error \geq 0.5$ **do**
  $finishTimes$ = executeTasks($blockSizeList$);
  synchronize();
  $blockSizeList$ = evaluateNextBlockSizes($finishTimes$);
  $fitValues$ = determineCurveProcessor();
end while
return $fitValues$;

Estes passos são mostrados no algoritmo 1. A variável $blockListSize$ contem o tamanhos dos blocos atribuídos a cada processador e é inicializada com $initialBlocoSize$,

que é definida pelo usuário. A variável $fitValues$ contem o resultado do ajuste por mínimos quadrados, incluindo o erro no ajuste, que é incializado como $+\infty$.

No laço inicia, enquano o erro é maior que um valor pré definido, a função envia um pedaço de dados para cada processador e obtem o tempo que demorou para realizar o processament em cada processador. Depois de esperar, que todos os processadores terminem, é determinado o tamnho do bloco para a próxima iteração, baseado no tempo de término de cada processador. Por fim, o algoritmo tenta ajustar um modelo de curva para cada processador e recebe o resultado do ajuste.

Seleção do tamanho de bloco ótimo: O algoritmo determina o tamanho do bloco ótimo para cada processador com o objetivo de minimizar o tempo total da aplicação. Considere que existem n processadores e o tamanho da entrada seja Z. O algoritmo atribui um pedaço de dados de tamnho x^g para cada processador $g = 1,...,n$, correpondendo a uma fração dos dados de entrada, tal que $\sum_{g=1}^{n} x^g = Z$. Denota-se como $E^g(x^g)$ o tempo de execução da tarefa E no processador g, para cada entrada de tamanho x^g. Para distribuir o trabalho entre os processadores, encontra-se um conjunto de valores:

$$X = \{x^g \in \mathbb{R} : [0,1] / \sum_{g=1}^{n} x^g = Z\} \tag{4.2}$$

que minimiza $E_1(x_1)$ enquanto satisfaz a restrição:

$$E_1 = E_2 = ... = E_n \tag{4.3}$$

que representa que todos os processadores gastariam uma mesma quantidade de tempo realizando o processamento. Para determinar o conjunto de valores de x, resolve-se o sistema de equações das curvas ajustadas para todos os processadores, dados por:

$$\begin{cases} E_1 = f(x_1) \\ E_2 = f(x_2) \\ E_n = f(x_n) \end{cases} \tag{4.4}$$

O sistema de equações é resolvido aplicando interior *point line search filter method*. O algoritmo busca minimozar as funções no espaço de busca. Ele alcança uma solução ótima percorrendo o interior de uma solução viável. Determinar o valor de x tal que o tempo é mínimo.

Rebalanceamento: Depois de resolver o sistema de equações o escalonador mantem enviando tarefas de tamanho x_i para cada processador i, logo que o processador termina a tarefa anterior. Também monitora o tempo de término de cada tarefa. Se a diferença no tempo de término entre x_i e x_j de qualquer duas tarefas i e j for maior que um limiar, o processo de balanceamento é reexecutado. Neste caso, o algoritmo

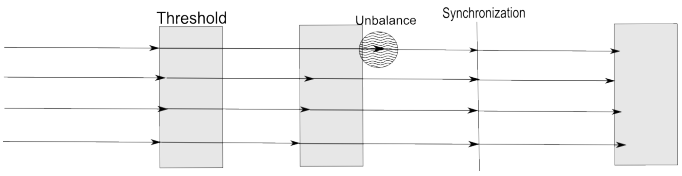

Figura 4.2: Execução das Threads com limiar

aplica o modelo de desempenho e a rotina de determinação do tamanho do bloco ótimo novamente. O escalonador então sincroniza as tarefas e inicia usando o novo valor de x_i. O limiar precisa ser determinado empiricamente através de alguns testes. Se o limiar é um valor muito pequeno, ocorre balanceamentos desnecessários, que aumentará o tempo total da aplicação. Se o limiar é muito grande balanceamento necessários podem não ocorrer, o que pode fazer com que threads fiquem ociosas.

A figura 4.2 apresenta um diagrama de execucão para quatro threads e uma thread desbalanceada. As caixas representam os limiares, as linas as threads e os círculos o desbalanceamento. As threads iniciam sincronizadas, recebem carga, mas em um segundo passo uma thread leva mais tempo que o valor de limiar, isto causa o desbalanceamento. Detectado o desbalanceamento, todas as threads são sincronizadas, e as partes são recalculadas para cada thread, fazendo com que as threads voltem ao estado sincronizado.

Algorithm 2 Complete dynamic algorithm

```
function dynamic()
fitValues = determineModel()
X = solveEquationSystem(fitValues);
while there is data do
    finishTimes = executeTasks(X);
    if maxDifference(finishTimes)≥ threshold then
        fitValues = determineCurveProcessor();
        X = solveEquationSystem(fitValues);
        synchronize();
    end if
end while
```

Algoritmo Completo: Algoritmo 2 apresenta o pseudocódigo do algoritmo de escalonamento. A função $determineModel$ mostrada no algoritmo 1 retorna o modelo de desempenho para cada processador. Então resolve o sistema de equações para determinar a melhor distribuição de pedaços de dados para cada processador.

O laço inicial repete enquanto há dados ainda para serem processados. É distribuído pedaços de dados para cada processador do sistema, obtendo o tempo de

término para cada execução de tarefa. É checado se a diferença maxima entre o término das tarefas está acima de um limar, obtido empiricamente. Se o limiar é alcançado, o algorimto ajusta um novo modelo de curvas e resolve o sistema de equações para estas novas curvas para determinar uma nova distribuição de tamanhos pedaços de dados.

4.2 IPOPT

IPOPT (Interior Point Optimizer) é um pacote de software aberto para otimização não linear. IPOPT pode resolver problemas de programação não-linear da seguinte forma:

onde x $\in$ a $\Re^n$ são as variaveis de otimização, $f : R^n emR$ é a função objetivo, e $g : R^n$ em R^M são as restrições não lineares. A função $f(x)$ e $g(x)$ podem ser linear ou não linear.

Para resolver um problema de otimização precisa-se criar um IpoptProblem com a função CreateIpoptProblem, que posteriormente precisa ser passado para a função IpoptSolve. O IpoptProblem criado por CreateIpoptProblem contem as dimensões do problema, as variaveis e os limites das restrições.

Capítulo 5

Implementação

5.1 Introdução

A aplicação foi feita na linguagem C, com o *framework* StarPU. Augonnet et al. (2010). StarPU é uma ferramenta para programação paralela que suporta arquiteturas híbridas como CPUs multicore e aceleradores. StarPU propõe uma abordagem de tarefas independentes baseada na arquitetura. Codelets são definidos como uma abstração de uma tarefa que pode ser realizada em um núcleo de uma CPU multicore ou submetido a um acelerador. Cada codelet pode ter várias implementações, um para cada arquitetura em que codelet pode ser realizada utilizando linguagens específicas e bibliotecas para a arquitetura alvo. Uma aplicação StarPU é descrita como um conjunto de Codelets com dependências de dados.

A ferramenta tem um conjunto de políticas de escalonamento implementadas que o programador pode escolher de acordo com as características da aplicação. A principal delas é a uso do algoritmo de escalonamento estático HEFT (Heteregeneous Earliest Finish Time) para agendar tarefas com base em modelos de custo de execução da tarefa.

Para cada dispositivo, como GPU ou CPU, foi programado um codelet. Um codelet é uma estrutura que representa um núcleo computacional. Tal codelet pode conter uma implementação do mesmo kernel em diferentes arquiteturas (por exemplo, CUDA e x86). As aplicações foram implementadas dividindo o conjunto de dados em tarefas. As tarefas são independentes, com cada tarefa a receber uma parte do conjunto de entrada.

Para avaliar o nosso algoritmo de balanceamento de carga, nós modificamos o algoritmo de escalonamento padrão do StarPU. A modificação do algoritmo de balanceamento de carga é feita alterando a variável STARPU_SCHED. O *framework* STARPU tem uma API que permite modificar as políticas de escalonamento. Existem estruturas de dados e funções que aceleram o processo de desenvolvimento. Por exemplo, a

função ”double starpu_timing_now (void)”que retorna a data atual em micro segundos, o que torna mais fácil para a determinação de medidas de tempo de execução.

Três outros algoritmos foram implementados para comparação: o guloso, o estático e HDSS. O guloso consistiu em dividir o conjunto de entrada em pedaços e atribuir cada pedaço da entrada a qualquer processador ocioso, sem qualquer atribuição de prioridade. O estático de Camargo (2012), mede as velocidades de processamento antes da execução e atribui um conjunto de blocos estático para cada processador, no início da execução, com os tamanhos do bloco proporcional à velocidade do processador. Por fim, o HDSS **?** foi implementado utilizando a estimativa dos mínimos quadrados para estimar os pesos e dividido em duas fases: fase de adaptação e fase de conclusão.

A biblioteca utilizada para resolver o sistema de equações é a IPOPT **?**. IPOPT (Interior Ponto Otimizar) é um pacote de software de código aberto para otimização não-linear em grande escala. Ele pode ser utilizado para resolver problemas de programação lineares gerais.

Tabela 6.1: Configuração das máquinas

Máquina	Modelo CPU	Número de GPUs	Memória da GPU	Modelo de GPU
A	Intel i7 a20	4 x GTX200b	896MB	GTX 295
B	Intel i7 4930K	2 x GK104	2GB	GTX 680
C	Intel i7 3930K	2 x GK110	6GB	GTX Titan

Capítulo 6

Resultados

6.1 Resultados dos experimentos

Foram usasdotrês diferentes máquinas para testar o algoritmo, apresentado na tabela **??**. Maquina A tem uma CPU i7 a20, com 4 cores com velocidade de clock de 2,67 GHz, 8192 MB de cache, 8 GB de RAM and 2 nVidia GTX 295, cada GTX 295 tem 280 cores por GPU, clock da memória de 999 MHz e tamanho da banda de memória de 223,8 GB/segundos com 2 GPUs em cada. Maquina B tem um i7 4930K CPU, com 6 cores, clock de 3,4 GHz, memoria cache de 12288 KB, 32 GB de RAM e 2 nVidia GTX 680, GTX 680 tem 1546 cores, 6 Gbps clock de memória e tamanho da banda de memória de 192,2 MHz. Máquina C tem um CPU i7 3939K, com 6 cores, velocidade de clock de 3,2 GHz, memória cache de 12288KB, 32 GB de RAM e 1 nVidia GTX Titan, GTX Titan tem 2688 cores, velocidade da memória de 6Gbps e tamanho da banda de memória de 223,8 GB/segundo. Foram considerados os cenários com apenas a máquina A, com a máquina A, e com a três máquinas (A, B e C). Os computadores estão conectados por uma rede Gigabit Ethernet. Foi usado o sistema operacional Ubuntu 12.04 e CUDA 5.5.

Para usar todos os n multiprocessadores da GPU, é necessário criar ao menos n blocks. Mais que isso, cada multiprocessador simultaneamente executa grupos (chamados de warps) de m threads de um unico bloco. e muitos warps estão presentes em cada GPU para o uso eficiente dos processadores.

Em todos os testes, usa-se todos os multiprocessadores das GPUs lançando kernels com k blocos com 1024 threads por bloco, onde k é o numero de processadores na

GPU. Para as GPUs usadas, k é 14, 8 e 30 na GTX Titan, GTX 680 e GTX 295 respectivamente. Para as CPUs, foram usados todos os cores das CPUs, lançando uma tarefa pro core.

6.2 Multiplicação de Matrizes

O algoritmo foi testatdo para a multiplicação de matrizes usando uma, duas e três máquinas e quatro diferentes algoritmos de escalonamento: (1) algoritmo proposto, (2) estatico, (3) HDSS, e (4) StarPU (greedy).

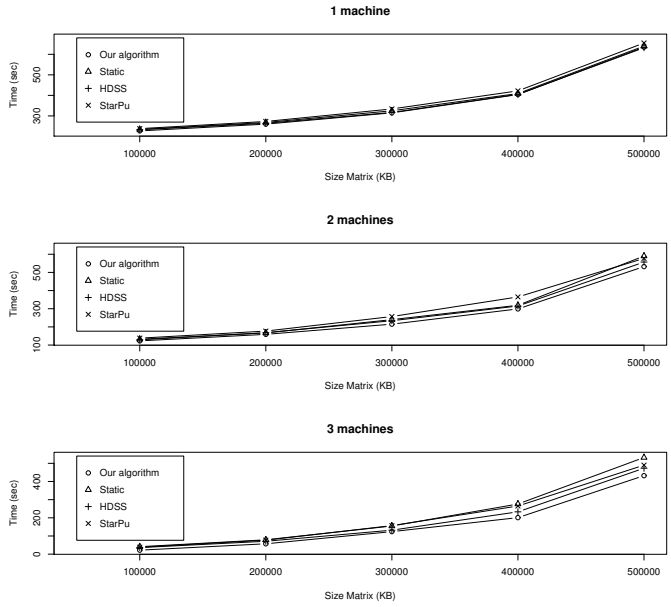

Figura 6.1: Diferença de execução com diferentes tamanhos de matrizes, para a multiplicação de matrizes

A figure 6.1 apresenta os resultados para matrizes com tamanhos 10.000 x 10.000 a 50.000 x 50.000. Em todos os cenários, o algoritmo de escalonamento proposto obteve melhores resultados, com o HDSS em segundo. O estático e o StarPU foram claramente inferiores.

Com uma máquina a diferença foi pequena porque há poucos tipos de dispositivos para o escalonador selecionar. Com duas máquinas nosso algoritmo começa a ter melhor desempenho, especialemente para casos de matrizes grandes, que é explicado pelo fato de que o tempo de execução da maultiplicação de matrizes aumenta rapidamente como aumenta-se o tamanho da matriz. Com três máquinas obteve-se o ambiente mais

heterogêneo e o ganho de desempenho usando o algoritmo proposto é o maior. Como nos outros cenários, aumentando-se o tamanho da matriz também aumenta-se o ganho de desempenho como o algoritmo proposto. Para matrizes 10.000 x 10.000, o algoritmo proposto gastou 20 segundos wnquanto o HDSS gastou 36, que resulta em 22,2% mais rápido. E para matrizes 50.000 x 50.000, o algoritmo proposto gastou 433 segundos enquanto o HDSS 473, que resulta em 8,49% mais rápido, o resumos está na tabela **??**

Tabela 6.2: Comparativo: HDSS x Algoritmo Proposto

	Tamanho Matriz 10,000 x 10,000 KB			tamanho Matriz 50,000 x 50,000		
Num. Máquinas	HDSS	Algoritmo Proposto	Differ. (%)	HDSS	Algoritmo Proposto	Differ. (%)
1 Machine	233.42	227.01	2.75	632.13	635.32	-0.50
2 Machines	129.43	123.62	4.49	557.92	532.11	4.63
3 Machines	36.85	22.43	39.13	473.64	433.41	8.49

Os resultados obtidos mostram que em ambientes mais heterogêneos o uso do algoritmo proposto apresenta mais vantagem. Esta vantagem é devido melhor distribuição dos dados ao longo da execução. A figura 6.2 mostra a diferença de tempo entre a primeira e a última thread, no cenário com três máquinas.

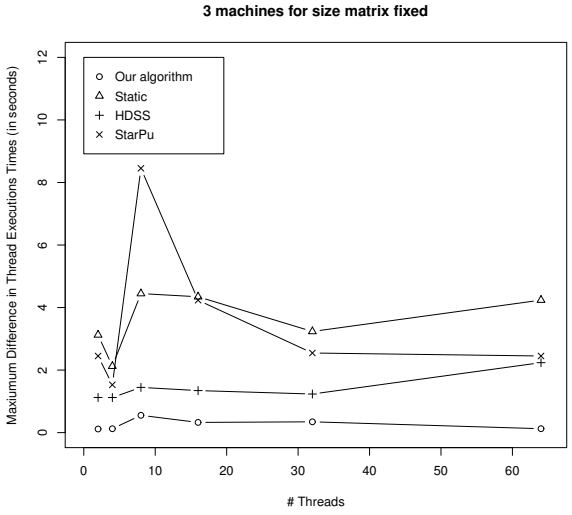

Figura 6.2: Diferença de tempo entre Threads

O algoritmo guloso do StarPU teve um bom desempenho, considerando que ele não tem diretamente a informação sobre a velocidade de processamento dos dispositivos. Mas ele usa estas informações indiretamente, desde que dispositivos mais rápidos que terminam suas tarefas mais cedo e, consequentemente, recebem mais tarefas. O algoritmo estático teve o pior desempenho, porque deixa a thread ociosa por muito tempo ver figura 6.2. Há grandes diferenças entre final de threads.

HDSS utiliza o início da execução para estimar o melhor tamanho de bloco e usa essa distribuição até o fim da execução. Além disso, grandes blocos são usados no início da execução, causando um atraso maior devido ao desbalanço quando usa diferentes tipos de dispositivos, como GPUs e CPUs. Por fim, HDSS utiliza uma aproximação bruta para a capacidade do dispositivo.

O algoritmo proposto tem bastante precisão na estimativa, a quantidade de dados que devem ser fornecida para cada unidade de processamento, a solução de um problema de optimização com o restrição de que todas as unidades devem terminar a execução das tarefas ao mesmo tempo. Quando a diferença entre terminar a execução de threads para certos partição de dados excede um certo limite, o algoritmo reequilibra os dados distribuição.

6.3 Blackscholes

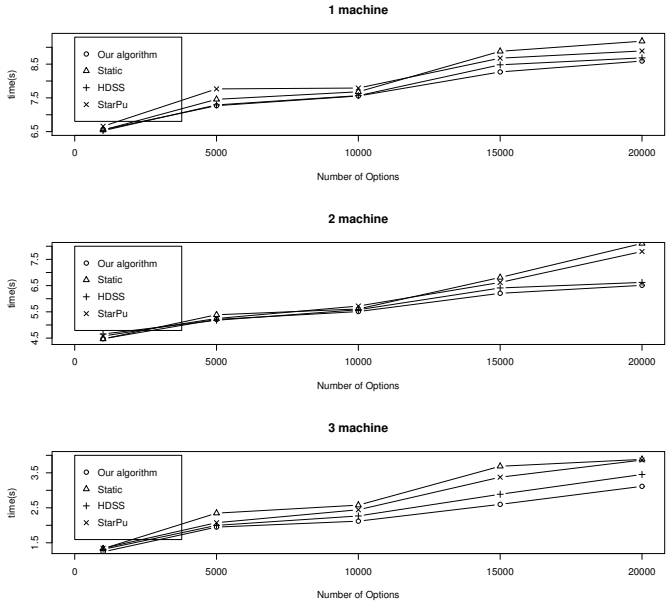

Figura 6.3: Diferença no tempo de execução para diferentes números de opção

Foram realizados testes de tempo de execução para a aplicação chamada blackscholes usando diferentes configuraçõs de máquinas. A figura 6.3 apresenta os tempo de execução, onde variou-se o número de opõs em cada execução e obteve-se o tempo de execução. Similarmente aos experimentos de multiplicação de matrizes, o melhor desempenho foi alcançado nos problemas maiores, com maior números de opções e

h

Tabela 6.3: Comparativo: HDSS x Algoritmo Proposto

	1000 Opções			20000 Opções		
Num. Máquinas	HDSS (s)	Algoritmo (s)	Diff. (%)	HDSS (s)	Algoritimo (s)	Diff. (%)
1	6.52	6.55	-0.45	8.68	8.59	1,03
2	4.65	4.47	3.87	6.62	6.51	1.66
3	1.31	1.24	5.34	3.45	3.11	9.81

em ambientes mais heterogêneos. Os resultados podem ser explicados com os mesmos argumentos. Blackscholes tem complexidade linear com o número de opções, mostrando que o algoritmo proposto é também últil para esta classe de problemas. A tabela 6.3 mostra os valores extremos com 1000 e 20000 opções, o melhor resultado foi com três máquinas, e com o maior número de opções.

Também, considerou-se que a aplicação termina em menos que 4 segundos, para o cenário com 3 máquinas, é possível notar que o overhead imposto pelo solucionador do sistema de equações para determinar a melhor distribuição é pequeno e o ganho obtido ultrapassa o custo dos cálculos. Para as aplicações testadas, em poucas iteracões, foi possivel obter a solução do sistema de equações, o que resultou em poucos milisegundos, na ordem de 10 ms.

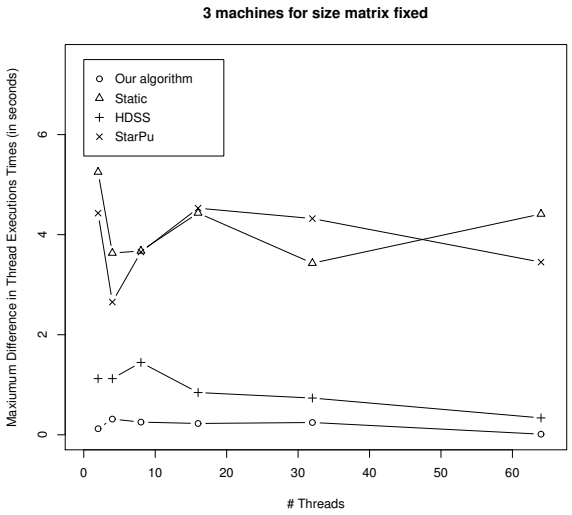

Figura 6.4: Time differences between the earliest and latest finishing threads

A figura 6.4 confirma o resultado de que a diferença entre a primeira a terminar e a última, no cenário com três máquinas é sempre menor para o algoritmo proposto.

6.4 Conclusões e Trabalhos Futuros

Foi proposto um algoritmo de escalonamento de tarefas para problemas de decomposição de domínio que executam em clusters de CPUs e GPUs hetererogêneas. O algoritmo proposto supera outros algoritmos similares devido a estimativa online da curva de desempenho para cada processador e a seleção da melhor distribuição para os dispositivos. foi apresentado para duas applicações que o algoritmo proposto fornece maiores ganhos para problemas grandes e em ambientes mais heterogêneos.

Embora usou-se clusters dedicados, pode-se também considerar o uso em núvem públicas, onde o usuário pode requisitar um numero de recursos alocados me máquinas virtuais de máquinas compartilhadas. Neste caso, a qualidade do serviço pode alterar durante a execução, e a adição de um limiar permite ajustar a distribuição de dados. Pode-se também considerar o ceneario com a adição de tolerância a falhas, onde máquinas podem ser tornar indisponíveis durante a execução. Neste cenário, uma simples redistribuição dos dados entre os dispositivos restantes permitiria que a aplicação ser readapte a este cenário.

Como trabalho futuro, espera-se incluir o custo da comunicação no algoritmo de escalonamento, que é essencial para aplicações onde o tempo gasto com a troca de informação entre tarefas não pode ser ignorada.

Capítulo 7

Plano de Trabalho

7.1 Introdução

Os detalhes do plano de tabalho são apresentados na tabela 7.1. O aluno ingressou no programa em fevereiro de 2013 e tem previsão de término em dezembro de 2014.

Etapa	2013											2014										
	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12
Disciplinas do Mestrado	X	X	X	X	X	X	X	X	X	X	X												
Levantamento Bibliográfico	X	X	X	X	X	X	X																
Desenvolvimento do Algoritmo					X	X	X	X	X	X	X	X	X	X									
Implementação do Algoritmo							X	X	X	X	X	X	X	X	X	X	X	X					
Implementação da Comunicação																			X	X	X		
Experimentos													X	X	X	X	X	X	X	X	X	X	
Escrita da Qualificação												X	X	X									
Escrita de Artigo																		X	X	X	X		
Escrita da Dissertação																				X	X	X	

Tabela 7.1: Cronograma das atividades previstas

Referências Bibliográficas

Acosta, A., Blanco, V., and Almeida, F. (2012). Towards the dynamic load balancing on heterogeneous multi-gpu systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 646–653.

Augonnet, C., Thibault, S., and Namyst, R. (2010). Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Rapport de recherche RR-7240, Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, http://hal.inria.fr/inria-00467677/PDF/RR-7240.pdf.

Ayguade, E., Badia, R., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J., and Quintana-Ortí, E. (2009). A proposal to extend the openmp tasking model for heterogeneous architectures. In Müller, M., Supinski, B., and Chapman, B., editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 154–167. Springer Berlin Heidelberg.

Borelli, F., Camargo, R., Martins, D., Stransky, B., and Rozante, L. (2012). Accelerating gene regulatory networks inference through gpu/cuda programming. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2012 IEEE 2nd International Conference on*, pages 1–6.

Camargo, E., Kostin, S., and Pinto, R. (2011). A tool for scientific visualization based on particle tracing algorithm on graphics processing units. In *Sistemas Computacionais (WSCAD-SSC), 2011 Simposio em*, pages 10–10.

Cederman, D. and Tsigas, P. (2008). On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. (2010). Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12.

CUDA Development Core Team (2012). CUDA 4.1 Programming Guide.

de Camargo, R. (2011). A multi-gpu algorithm for communication in neuronal network simulations. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10.

de Camargo, R. (2012). A load distribution algorithm based on profiling for heterogeneous gpu clusters. In *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, pages 1–6.

Diamos, G. F. and Yalamanchili, S. (2008). Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA. ACM.

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 47–, Washington, DC, USA. IEEE Computer Society.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA. ACM.

Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

Guevara, M., Gregg, C., Hazelwood, K., and Skadron, K. (2009). Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, PMEA, pages 69–76, Raleigh, NC.

Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA. ACM.

Leung, J. Y. T., editor (2004). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Computer and Information Science Series. Chapman & Hall/CRC Press, Boca Raton, FL, USA, 1 edition.

Li, Q., Salman, R., Test, E., Strack, R., and Kecman, V. (2011). Gpusvm: a comprehensive cuda based support vector machine package. *Central European Journal of Computer Science*, 1(4):387–405.

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. (2008). Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296.

M. Mller, C., Strengert, M., and Ertl, T. (2007). Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Comput.*, 33(6):406–419.

Miyoshi, T., Irie, H., Shima, K., Honda, H., Kondo, M., and Yoshinaga, T. (2012). Flat: a gpu programming framework to provide embedded mpi. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 20–29, New York, NY, USA. ACM.

Owens, D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(1):879–899.

Riegel, E., Indinger, T., and Adams, N. A. (2009). Implementation of a lattice–boltzmann method for numerical fluid mechanics using the nvidia cuda technology. *Computer Science - Research and Development*, 23(3-4):241–247.

Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M., Meira, W., Catalyurek, U., and Ferreira, R. (2009). Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10.

Wang, L., Huang, M., Narayana, V. K., and El-Ghazawi, T. (2011a). Scaling scientific applications on clusters of hybrid multicore/gpu nodes. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 6:1–6:10, New York, NY, USA. ACM.

Wang, L., Jia, W., Chi, X., Wu, Y., Gao, W., and Wang, L.-W. (2011b). Large scale plane wave pseudopotential density functional theory calculations on gpu clusters. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10.