Escrevendo e Paralelizando Programas - Estudo de Caso em Algoritmos Genéticos

Resumo: Este estudo pretende desmitificar que desenvolvimento de programas utilizando computação paralela é tarefa difícil e complicada, através da apresentação de estudo de caso com Algoritmos Genéticos. O assunto deve interessar aqueles que trabalham com algoritmos de otimização evolucionários que ainda resistem em não empregar paralelismo. O texto percorre inicialmente conceitos básicos de Algoritmos Genéticos e Computação Paralela, depois aprofunda-se nos detalhes de implementação computacional, e ao final, descreve o experimento e discute os resultados de medidas de desempenho.

# Introdução

Em engenharia, otimizar significa buscar pelo melhor conjunto de parâmetros que modelam matematicamente um problema através do emprego de técnicas, métodos ou algoritmos de otimização. Considerando um dado equipamento, as técnicas de otimização poderiam ser empregadas para maximizar desempenho e minimizar custos, que neste caso representam as funções objetivo. Em problemas reais, as funções objetivo podem envolver diferentes variáveis (de natureza contínua, discreta ou mista) e resultar, por exemplo, em funções descontínuas, não-lineares, não-convexas e multi-modal. Esses “dificultadores” podem caracterizar o problema de otimização como Não-Polinomiais Completos (NPC); classe de problemas onde o tempo de execução é exponencial em função do número de variáveis da entrada, sendo inviável a pesquisa pela solução por avaliação de todas as combinações. Então, para se obter resultados para NPCs, pesquisadores tais como [x], [y] e [z], empregam heurísticas, como o Algoritmos Genéticos (AGs). Mesmos os AGs despendem custo computacional para tratar os NPCs, isto motiva aplicação de computação paralela para acelerar a resolução do problema.

A programação paralela permite que partes concorrentes de um algoritmo sejam executadas simultaneamente, utilizando melhor dos recursos do processador e reduzindo o tempo de processamento. Além da aplicação ser paralelizável ainda há duas condições essenciais à programação paralela, o hardware e a linguagem de programação utilizados devem suportar este modelo de programação. Estas condições são facilmente atendidas considerando-se que todos computadores novos comercializados atualmente possuem suporte à programação com múltiplas threads (linhas de execução) e existem bibliotecas e APIs, tais como OpenMP, MPI, PThread, para diversas linguagens de programação (C/C++, Java, Fortran, entre outras) que permitem a criação de threads em um código já existente. Uma das bibliotecas que possuem maior facilidade de aprendizado é o OpenMP, utilizado neste capítulo para criação das threads em C++.

Os AGs possuem métodos de evolução, avaliação e seleção que podem ocorrer simultaneamente em determinados trechos, portanto podem obter redução no tempo de execução (Speedup) com programação paralela. Em [inicioAGP] foi proposto o Algoritmo Genético Paralelo (AGP) e desde então surgiram diversas propostas de modelos de programação paralela para o AGP, tais como modelo de ilha e inserção do paralelismo nos métodos evolutivos, onde a evolução e avaliação da população anterior sejam simultâneos. Existem ainda diversos outros modelos, os quais possuem características que podem promover ganho para aplicações específicas.

# Revisão

## Algoritmos Genéticos

Os AGs são técnicas de otimização projetadas em observação a teoria da evolução proposta por Darwin. Nessa analogia aos sistemas naturais, um indivíduo **x^i** é uma possível solução para o problema. Esse indivíduo participará, dependendo de seu desempenho, de uma ou mais gerações (iterações) do processo evolutivo (otimização). A função de desempenho ou função objetivo f(**xi**) simula o habitat onde o indivíduo se encontra e fornece como resultado uma nota de aptidão d\_**x^i** que é empregada no mecanismo de seleção. O método de seleção, por sua vez, busca priorizar a escolha de indivíduos com “boas” notas a cada geração. O grupo selecionado é submetido aos processos de cruzamento e de mutação, para troca de informações e inserção de diversidade respectivamente. O processo evolutivo continua ciclicamente (avaliação, seleção, cruzamento, mutação) até algum critério de parada seja atingido. Trabalhando com populações de indivíduos, os AGs tem condições de extrair e combinar informações existentes de forma eficiente. A Tabela 1 detalha uma estrutura algorítmica típica de um AG com funcionalidades básicas, e a Tabela 2 descreve os símbolos da Tabela 1.

Tabela 1: Algoritmo AG com funcionalidades básicas.

|  |
| --- |
| Entradas: f(**x**), **D**(**x**)  Saída: (**x\*,** f(**x\***))  1- Defina métodos Seleção, Cruzamento e Mutação.  2- Defina valores para n\_pop, p\_c e p\_m.  3- Defina codificação do indivíduo.  4- Defina o critério de parada.  5- Inicialize população pop\_x.  6- Se critério de parada é atingido, vá para 15.  7- Avalie população com f(**x**); armazene o desempenho em des\_x.  8- Armazene o arg max des\_x em **x\***.  9- Execute Seleção(pop\_x,des\_x); armazene resultados em pop’\_x e des’\_x  10- Execute Cruzamento(pop’\_x,p\_c); armazene resultado em pop’\_x.  11- Execute Mutação(pop’\_x,p\_m); armazene resultado em pop’\_x.  12- Escolha aleatoriamente **i**; substitua **x^i** por **x\*** e atualize pop’\_x e des’\_x.  13- Copie pop’\_x em pop\_x e des’\_x em des\_x.  14- Vá para 6.  15- Retorne **x\*** e f(**x\***). |

Tabela 2: Entidades e estrutura de dados AG (Tabela 1).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entidade | Sistemas Naturais | Sistemas Artificiais | Tipo | Dimensão |
| pop\_x | população | soluções candidatas | matriz | n\_pop x IR^n |
| des\_x | desempenho da população | valor da função objetivo | vetor | n\_pop |
| **x^i** | indivíduo i | solução candidata i | vetor | IR^n |
| **x\*** | indivíduo de mais apto no ambiente. | ponto ótimo, solução do problema | vetor | IR^n |
| **D**(**x**) | fronteiras do habitat | Domínio das soluções candidatas | vetor intervalar | IR^n |
| f(**x**) | avaliação do indivíduo no habitat | função objetivo | função | IR^n->IR |
| n\_pop | tamanho da população | número de soluções candidatas | constante | IR |
| p\_c | probabilidade de ocorrência de cruzamento | probabilidade de recombinação | constante | IR |
| p\_m | probabilidade de ocorrência de mutação | probabilidade de inserção de diversidade | constante | IR |

Os AGs se popularizaram no mundo científico devido a diversas características, dentre as quais destacam-se a:

(a) independência de continuidade e do cálculo de derivadas.

(b) *robustez* a variações do problema tratado, pois a calibração do conjunto de parâmetros ajusta o comportamento do AG da função objetivo, sem necessidade de alterar código;

(c) *flexibibilidade* a resolução de diferentes problemas, pois com alterações relativamente simples no código o AG fica adaptado para tratar outras classes de problemas; e

(d) *eficácia* em encontrar a região de otimalidade, pois trabalhando com populações a recombinação dos melhores favorecem a evolução para região onde se encontra ponto ótimo.

Considerando (I) as características enumeradas anteriormente, os AGs apresentam-se como interessantes opções para tratar problemas NPC e (II) que os AGs empregam populações e há diferentes formas de paralelizar seus métodos internos, decidiu-se estudar o comportamento de um AG paralelo frente a um NPC.

## Tecnologias para Computação Paralela

A programação paralela pode ser realizada através de técnicas de memória distribuída (cluster), onde há diversos processadores conectados por rede que acessam sua própria memória, ou técnicas de memória compartilhada, onde diversos núcleos de um único processador acessam a mesma memória.

Os computadores comercializados atualmente possuem suporte à programação paralela com memória compartilhada, sendo classificados como MIMD (Multiple Instruction Multiple Data) segundo a taxonomia de Flynn [Flynn:1972], e existem diferentes configurações para tais máquinas.

Os computadores que mais têm se destacado comercialmente possuem processadores multicore, ou seja, com mais de um core (núcleo), onde os cores podem ser fisicamente existentes e independentes, ou parte destes cores é apenas simulada pelos cores reais, o que pode levar a ganho de desempenho em relação a um processador que tenha metade do número de núcleos deste ou perda de desempenho em relação a um processador de mesmo número de cores fisicamente existentes, devido à concorrência por recursos. Esta técnica de simulação é chamada de Simultaneous Multithreading (SMT) e foi adotada pela Intel para seus processadores com o nome Hyper-Threading (HT).

Enquanto uma CPU (Central Processing Unit) possui alguns núcleos, uma GPU (*graphics processing unit*) pode possuir dezenas, ou centenas, vezes mais núcleos em uma placa de vídeo, sendo núcleos menores e com menores funcionalidades que os núcleos da CPU, porém com a capacidade de realizar grandes quantidades de cálculos simultaneamente. A GPU foi criada com o propósito de ser utilizada apenas para criação de imagens, devido à sua capacidade de operar grandes quantidades de números de ponto-flutuante e sua restrita funcionalidade a estes. Porém, em 2000 alguns pesquisadores começaram a utilizar a recém-lançada GPU da NVIDIA, GeForce 256, para programação de propósito geral [Trendall], surgindo então o conceito General Purpose GPU (GPGPU).

Novas arquiteturas foram lançadas integrando GPU e CPU, a APU (*Accelerated Processing Units*), em uma tentativa de reduzir um grande problema na GPGPU, o tempo de acesso da GPU à memória principal. Porém ainda permanece a dificuldade para a nova forma de desenvolvimento.

Desenvolver um programa que utilize tantos núcleos simultaneamente pode produzir excelentes resultados, entretanto não é trivial a elaboração de um código que comunique diretamente com este hardware. Inicialmente se utilizava das próprias linguagens para criação de imagens para tanto, como o OpenGL, mas além da dificuldade para desenvolvimento havia limitações que impediam a programação de determinados programas mais complexos. Surgiram então compiladores que eram extensões de linguagem Fortran e C, que passaram a facilitar, e em alguns casos viabilizar, o desenvolvimento.

Mesmo utilizando estes compiladores são necessários conhecimentos a respeito do hardware disponível e é necessário a alteração do código para adaptação à GPU. Portanto, como este capítulo tem o objetivo da introdução à programação paralela, não será utilizada a GPGPU, que poderia tornar este estudo introdutório uma barreira a iniciação desta forma de programar.

**Memória compartilhada**

Plataformas SMT, máquinas multicore, e computadores de memória compartilhada proveem suporte ao sistema de execução de múltiplos fluxos de instruções independentes, ou seja, threads. Porém, sem o bom uso por meio dos programas que levem a arquitetura do computador em consideração estas threads não serão bem utilizadas, e o tempo de processamento de um programa poderá ser semelhante ao tempo de execução em um computador com apenas um core [Henessy].

Com a abordagem de memória compartilhada há a necessidade de troca de informação entre os cores para que o processamento seja distribuído entre as threads.

Diferentes cores dentro de um mesmo chip possuem acesso à mesma memória global e cada um possui acesso à uma memória mais rápida porém individual, a memória cache. Quando um programa utiliza uma variável de escopo global, acessada por mais de uma thread, é necessário que as threads sejam sincronizadas para evitar que uma acesse o valor antigo, ou seja, gravado pela outra thread, da variável.

É esperado que um programa obtenha o mesmo comportamento tanto sequencial quanto paralelamente, o paralelismo deve apenas permitir que a execução seja mais rápida. Para isto é necessário que as variáveis mantenham seu valor de forma independente para cada thread, ou seja, cada thread possui sua cópia da variável, para que a semântica do processamento seja mantida. Mas, o comportamento de uma execução do programa pode gerar dados que outra execução não geraria, devido ao acesso aleatório das threads aos dados, portanto, o procedimento não é exatamente reprodutível, apesar de apresentar o mesmo comportamento. [UsingOpenMP]

Quando o desenvolvedor utiliza o OpenMP, é esta biblioteca a responsável por todo esse gerenciamento, bastando ao desenvolvedor informar quais dados devem ser privados à cada thread, e informar quando as threads devem atualizar o valor na memória global e esperar umas pelas outras.

**Decisões de Projeto**

Cada modelo de hardware (arquitetura de computador) possui características que podem influenciar na escolha da linguagem de programação e API a se utilizar para o paralelismo, conforme apresentado em [Hennesy]. Para um processador de 4 núcleos reais, possivelmente a programação com PThread pode obter maior Speedup, enquanto para um processador de 4 núcleos onde apenas 2 núcleos existem fisicamente o OpenMP pode obter melhores resultados. Porém, não há uma restrição apenas com relação à arquitetura, mas também em relação à carga de trabalho e a forma de implementação.

Existem diversos modelos para criação do paralelismo no AG, porém estes modelos devem, idealmente, levar em consideração a arquitetura utilizada. Se houver a disponibilidade do uso de um cluster de computadores é possível que o AGP com modelo de ilhas implementado em MPI obtenha maior speedup que outras implementações com programação de memória compartilhada. Entretanto, se houver disponibilidade de apenas um processador multicore, possivelmente esta implementação não será a mais eficiente.

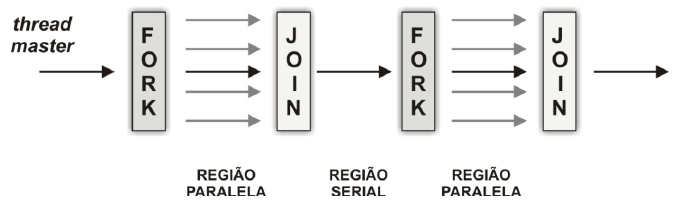
Para este capítulo optou-se pelo uso do OpenMP devido a este ser mais facilmente implementado, reduzindo a complexidade na programação, já que o objetivo deste capítulo é a introdução do uso de paralelismo, não a melhor forma de realizá-lo. Além de que com o uso do OpenMP há flexibilidade com relação à plataforma utilizada, permitindo até mesmo uma execução sequencial do programa, ou com múltiplas threads em um processador de único núcleo, o que é possível via mecanismo de troca de contexto.

**Modelo de programação em OpenMP**

OpenMP (Open Multi-Processing, ou Multi-processamento aberto) é baseado em modelo de memória compartilhada, o que implica em compartilhamento das variáveis, por padrão. Entretanto, algumas variáveis precisam que seu valor seja exclusivo à cada thread, ou seja, essa variável pode possuir diferentes valores, para cada thread, e por isso é chamada private.

A execução do processo em OpenMP inicia com apenas uma thread, que ao encontrar uma região paralela cria o número de threads determinado através do método disponibilizado pela biblioteca, o **omp\_set\_num\_threads**. Então essas threads executam simultaneamente até encontrar o fim desta região, onde apenas uma thread continuará a execução. Este processo pode ser repetido caso esta thread encontre novamente uma região paralela ou encerrado sequencialmente, caso contrário. Esse modelo de programação é conhecido como fork-join e pode ser ilustrado na Figura 1.

Figura 1 - Modelo fork-join



Tanto as regiões paralelas quanto as barreiras para sincronização são identificadas pelo compilador através de diretivas do OpenMP. Estas diretivas são “processadas” em tempo de compilação e determinam como o processador se comportará ao executar o código.

Além destas diretivas de compilação, o OpenMP é constituído por mais dois elementos. O primeiro é um conjunto de rotinas de biblioteca, como o método omp\_set\_num\_threads() que permite alterar o número de threads que o OpenMP criará ao encontrar uma região paralela. O outro elemento são as variáveis de ambiente, por exemplo a variável **OMP\_NUM\_THREADS** que setado como número de threads através do omp\_set\_num\_threads().

# Algoritmo Genético Paralelo

## Pseudocódigo

O AGP possui pseudocódigo semelhante ao AG, apenas incluindo a separação e sincronização das threads, como pode ser observado na Tabela 3. Cada thread terá sua própria cópia da população auxiliar e notas da população auxiliar, acessando a população global juntamente com as demais threads para realizar avaliação, seleção e evolução (cruzamento e mutação) paralelamente.

Tabela 3: Algoritmo AG com funcionalidades básicas.

|  |
| --- |
| Entradas: f(**x**), **D**(**x**)  Saída: (**x\*,** f(**x\***))  1- Defina métodos Seleção, Cruzamento e Mutação.  2- Defina valores para n\_pop, p\_c e p\_m.  3- Defina codificação do indivíduo.  4- Defina o critério de parada.  Para cada thread:  5- Inicialize população pop\_x.  6- Se critério de parada é atingido, vá para 15.  7- Avalie população com f(**x**); armazene o desempenho em des\_x.  8- Armazene o arg max des\_x em **x\***.  9- Execute Seleção(pop\_x,des\_x); armazene resultados em pop’\_x e des’\_x  10- Execute Cruzamento(pop’\_x,p\_c); armazene resultado em pop’\_x.  11- Execute Mutação(pop’\_x,p\_m); armazene resultado em pop’\_x.  12- Escolha aleatoriamente **i**; substitua **x^i** por **x\*** e atualize pop’\_x e des’\_x.  13- Copie pop’\_x em pop\_x e des’\_x em des\_x.  Sincronize as threads.  14- Vá para 6.  15- Retorne **x\*** e f(**x\***). |

## Código

Através da implementação inicial em C/C++ sequencial, podemos começar a inserção do OpenMP através da inclusão do omp.h: **#include <omp.h>**.

Para que então o compilador seja capaz de interpretar as diretivas, rotinas de biblioteca e variáveis de ambiente do OpenMP.

Para esta implementação foi acrescentada a variável global inteira **num\_th**, para determinar o número de threads da execução. Esta variável foi passada como parâmetro para o método **omp\_set\_num\_threads(num\_th);** no **main** do código.

Cada thread deve possuir sua própria população e população auxiliar, bem como os vetores com as notas de respectivas populações. Para tanto, a alocação de tais vetores deve ocorrer dentro da região paralela, porém utilizando estes como private será inviabilizado o seu uso em região sequencial posterior à região paralela. A variável do melhor indivíduo já encontrado deve ser acessada por todas as threads, já que elas irão procurar por diferentes espaços da região de solução e um delas pode encontrar um indivíduo melhor que o já encontrado até então, dentre todas as threads. Portanto, a alocação do vetor com o melhor indivíduo deve ser feita antes da região paralela, para que haja apenas uma cópia deste, e ele possa ser acessado posteriormente à região paralela.

A partir desta seção onde as variáveis já estão setadas, as threads podem executar concorrentemente sem esperar umas pelas outras, desta forma a diretiva para execução paralela do loop interno onde ocorrem evoluções e avalição da população podem ter o acréscimo **nowait**, para informar que ao fim do loop não há necessidade de o compilador criar uma barreira.

#include <omp.h> (1)

...

int num\_th = 4; // Número de threads (2)

…

int main(int argc, char \*argv[])

{

omp\_set\_num\_threads(num\_th); (3)

melhorInd = (int\*) malloc(num\_var \* sizeof(int));

#pragma omp parallel private(i, j, mInd, pInd, popAux, notasAux, populacao, notas) (4)

{

populacao = (int\*)malloc(tamanhoPopulacao\*num\_cid\*sizeof(int)); (5)

notas = (float\*)malloc(tamanhoPopulacao\*sizeof(float));

…

#pragma omp for schedule(static) nowait (6)

for(geracao = 1; geracao < maxGen; geracao++){

… (7)

}

}

}

# Problema de Otimização

O Problema do Caixeiro Viajante (PCV) é definido como o problema de um caixeiro para determinar a menor distância para se percorrer N cidades, sem repetição de cidade, retornando à cidade inicial, ou seja, ele pode ser modelado através de um grafo hamiltoniano. Sua solução ótima só é garantida através de algoritmo força bruta, o que torna impraticável o seu uso para um número muito grande de cidades sem nenhuma forma de otimização.

Este problema possui complexidade fatorial, e a adição de cada cidade na carga de trabalho pode aumentar fatorialmente o tempo de execução, inviabilizando seu uso. Por isto o PCV é frequentemente utilizado para validação de heurísticas, como o Algoritmo Genético (AG), que podem tornar viável o uso de grandes cargas de trabalho, porém não possuindo garantia de obter o valor ótimo.

# Experimentos e Resultados

Para aquisição de resultados iniciais e testes da qualidade do algoritmo, a princípio foram utilizadas cargas de trabalho que simulam cidades dispostas em círculo, pois assim é matematicamente garantido que o trajeto ótimo deverá ser menor que o comprimento da circunferência, bem como já se sabe o trajeto ótimo que é a ordem sequencial, crescente ou decrescente, das cidades, independentemente de seu ponto de partida.

Quanto maior o número de cidades dispostas no círculo mais próximo o valor do trajeto ótimo fica do valor de comprimento da circunferência. Porém, com esta carga maior, aumenta-se a dificuldade do algoritmo encontrar o trajeto ótimo através dos métodos de seleção e evolução adotados, uma vez que o cruzamento é realizado de forma que não há troca de informação entre diferentes indivíduos e a população inicial é gerada de forma aleatória sem qualquer restrição além das impostas pela definição do PCV.

Esta carga foi gerada através de um método que calcula as coordenadas das cidades no plano cartesiano pelo raio e número de cidades informado, como apresentado no Algoritmo [geracaoCidades]. Assim, foi possível realizar testes com diversos números de cidades para testar a estabilidade do algoritmo e detectar possíveis falhas ou fraquezas nos métodos evolutivos.

|  |
| --- |
| Entradas: raio, numeroCidades  Saída: estrutura de dados com coordenadas das cidades    Calcule deltaAlfa, tal que lhe seja atribuído 2\*pi/numeroCidades;  PARA i← 0 até numeroCidades, FAÇA:  Calcule alfa, tal que seja atribuído i \* deltaAlfa;  Insira na estrutura de dados utilizada a longitude ← raio \* cosseno(alfa), e latitude ← raio \* seno(alfa);  FIM PARA  Retorne a estrutura de dados com as coordenadas das cidades |

**Algoritmo [geracaoCidades]: Geração das cidades de entrada dispostas em círculo**

Código em C++:

float deltaAlfa = 2\*M\_PI/num\_cid, alfa;  
 for(int i=0; i<num\_cid; i++){  
 alfa = i\*deltaAlfa;  
 cidade[i].longitude = raio\*cos(alfa);   
 cidade[i].latitude = raio\*sin(alfa);  
 }

### Resultados

A Lei de Amdahl [Amdahl] determina o máximo que um sistema pode obter de speedup através da melhoria de um trecho deste. Com base na porcentagem do programa que pode ser paralelizado é determinado o valor máximo de speedup, independente do número de processos paralelos que possam ser acrescentados.



**Figura [Amdahl-Fig] - Lei de Amdahl**

Segundo a Lei de Amdahl, o ganho de desempenho que pode ser obtido em um sistema está relacionado com o tempo gasto com a melhoria sobre o tempo que essa parte do sistema é utilizada durante a sua operação. Numa situação hipotética, caso um programa possa ser paralelizado de forma que o tempo de processamento da região paralela tenda a zero, o tempo total de execução será limitado pelo tempo da parte sequencial.

O ideal é que o speedup seja a divisão do tempo de execução sequencial pelo tempo de execução paralela, de forma a obter o número de elementos de processamento disponíveis. Diversos fatores influem sobre o tempo de execução paralela, fazendo com que o speedup ideal não seja alcançado (speedup sub-linear) ou superado (speedup super-linear).

**Resultados carga inicial**

Com a carga gerada através das cidades dispostas em círculo foram obtidos resultados ...

**Resultados carga TSPLIB**

Após os resultados inicialmente obtidos, foi necessário apurar a robustez do algoritmo através de outro benchmark, que explore diferentes características através da carga de trabalho e teste diferentes combinações possíveis de parâmetros. Então, foram utilizadas cargas de trabalho do PCV disponibilizados na Traveling Salesman Problem Library (TSPLIB) [tsplib], onde há diversos testes com diferentes números de cidades e diferentes características para o trajeto ótimo, também disponibilizado pela TSPLIB.

# Referências

@book{navauxarquiteturas,

title={ARQUITETURAS PARALELAS},

author={NAVAUX, P.O.A. and ROSE, C.A.F.},

isbn={9788577803095},

publisher={BOOKMAN COMPANHIA ED}

}

@book{UsingOpenMP,

author = {Chapman, Barbara and Jost, Gabriele and Pas, Ruud van der},

title = {Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)},

year = {2007},

isbn = {0262533022, 9780262533027},

publisher = {The MIT Press},

}

@ARTICLE{tsplib,

author = {Reinelt, G},

title = {TSPLIB - A Traveling Salesman Problem Library},

journal = {ORSA Journal on Computing},

year = {1991},

volume = {3},

pages = {376--384},

number = {4},

url = {http://joc.journal.informs.org/cgi/content/abstract/3/4/376}

}

@ARTICLE{Flynn:1972,

author = {Flynn, Michael J.},

title = {Some computer organizations and their effectiveness},

journal = {IEEE Trans. Comput.},

year = {1972},

volume = {21},

pages = {948--960},

number = {9},

month = sep,

acmid = {1952459},

address = {Washington, DC, USA},

doi = {10.1109/TC.1972.5009071},

issn = {0018-9340},

issue\_date = {September 1972},

keywords = {Computer organization, computer organization, instruction stream,

overlapped, parallel processors, resource hierarchy},

numpages = {13},

publisher = {IEEE Computer Society},

url = {http://dx.doi.org/10.1109/TC.1972.5009071}

}

@BOOK{Rauber2010,

title = {Parallel Programming for Multicore and Cluster Systems},

publisher = {ACM},

year = {2010},

editor = {Springer},

author = {Thomas Rauber and Gudula Rünger}

}

@inproceedings{Amdahl,  
 author = {Amdahl, Gene M.},  
 title = {Validity of the single processor approach to achieving large scale computing capabilities},  
 booktitle = {Proceedings of the April 18-20, 1967, spring joint computer conference},  
 series = {AFIPS '67 (Spring)},  
 year = {1967},  
 location = {Atlantic City, New Jersey},  
 pages = {483--485},  
 numpages = {3},  
 url = {http://doi.acm.org/10.1145/1465482.1465560},  
 doi = {10.1145/1465482.1465560},  
 acmid = {1465560},  
 publisher = {ACM},  
 address = {New York, NY, USA},  
}

@inproceedings{Trendall,

author = {Trendall, Chris and Stewart, A. James},

title = {General Calculations using Graphics Hardware with Applications to Interactive Caustics},

booktitle = {Proceedings of the Eurographics Workshop on Rendering Techniques 2000},

year = {2000},

isbn = {3-211-83535-0},

pages = {287--298},

numpages = {12},

url = {http://dl.acm.org/citation.cfm?id=647652.732131},

acmid = {732131},

publisher = {Springer-Verlag},

address = {London, UK, UK},

}