UMA ABORDAGEM *BRANCH AND BOUND* PARA O RCPSP EM UM AMBIENTE DE COMPUTAÇÃO COLABORATIVA

Fábio C. Lourenço*, Felipe C. Pereira, Éber A. Schmitz
Núcleo de Computação Eletrônica / Instituto de Matemática - UFRJ, Rio de Janeiro – Brasil
fabiocl@ufrj.br*, felipep@ufrj.br, eber@nce.ufrj.br

Felipe M. G. França
PESC – COPPE - UFRJ, Rio de Janeiro – Brasil
felipe@cos.ufrj.br

RESUMO: Um projeto pode ser representado por uma rede de atividades formando um grafo de precedência, direcionado e acíclico. Quando a quantidade de recursos existentes é limitada, o problema de determinação do menor tempo de realização do projeto é conhecido como RCPSP (Resource Constrained Project Scheduling). A solução ótima para o problema RCPS é reconhecidamente NP-hard. Este artigo mostra um algoritmo distribuído para a solução ótima do problema RCPS usando uma abordagem branch and bound. Este algoritmo foi implementado e avaliado num ambiente de computação colaborativa, do tipo peer to peer, com escalonamento adaptativo distribuído para balanceamento de carga nos nós computacionais. Os resultados sugerem a escalabilidade do algoritmo apenas com a adição de nós computacionais.

Palavras chaves: RCPSP, branch and bound, computação colaborativa, paralelismo.

1. Introdução

Muitos trabalhos de pesquisas já foram realizados na área de escalonamento de atividades de projetos (Fleszar e Hindi, 2002; Herroelen *et. al.*, 1997). Estes trabalhos tinham como foco principal a modelagem e o desenvolvimento de algoritmos nessa área. Neste artigo, apresentamos uma nova abordagem ao problema do escalonamento ótimo do RCPSP (*Resource-Constrained Project Scheduling Problem,* Demeulemeester, *et. al.,* 1999) utilizando uma arquitetura de processamento colaborativo. O problema de encontrar o escalonamento ótimo do RCPSP visando a minimização do tempo total (*makespan*) é da classe *NP-hard* (Garey e Johnson, 1979).

O RCPSP é caracterizado pela existencia de restrições de precedência, que determina o encadeamento de atividades obrigando que umas terminem antes que outras comecem; e restrições de recursos, que limitam a quantidade de recursos que podem ser utilizados nas execuções das tarefas por unidade de tempo.

Um dos pontos principais do algoritmo apresentado neste artigo é a utilização de uma arquitetura de computação colaborativa (Sarmenta, 2001) para paralelizar a busca pela solução ótima. A arquitetura de computação colaborativa simula uma máquina paralela com troca de mensagens usando um conjunto de computadores *desktop* interconectados – que chamaremos de nós.

Para o uso eficiente da arquitetura colaborativa é necessário solucionar problemas secundários: um que trata da divisão do problema em subproblemas menores (pacotes de trabalho), chamado de problema de particionamento; e outro que trata da alocação de cada um dos pacotes de trabalho para os nós, que chamaremos de problema de alocação

A principal causa do problema de particionamento é a diferença entre a taxa de transferência do meio de interconexão dos nós e sua própria velocidade de processamento. Em outras palavras, o tempo que cada nó leva para resolver um pacote deve ser tal que compense o tempo gasto para na transmissão (Shread, 2002).

Outro cuidado importante é na geração e distribuição de trabalho entre os nós que participam da solução de um problema. Uma estratégia bastante utilizada para aplicações com este perfil é a

utilização do modelo *master-worker* (Sarmenta, 2001). Neste modelo, há um nó (*master*) responsável por gerar e distribuir trabalho entre os nós que irão realizar o processamento (*workers*). A distribuição de pacotes de trabalho no modelo *master-worker* geralmente é feito por demanda, ou seja, um nó se registra como *worker* junto ao *master* e recebe deste um pacote de trabalho. Ao entregar a resposta do pacote, o nó *master* despacha um novo pacote de trabalho para o *worker*. Desta maneira, naturalmente, os nós com maior poder computacional receberão mais pacotes de trabalho do que nós menos potentes, atingindo um bom balanceamento.

2. Revisão dos procedimentos de solução para o RCPS

Devido à complexidade do RCPSP, a maioria das tentativas de solução adotam heurísticas para obter uma solução aproximada. Dentre as principais abordagens para solução aproximada tendo o *makespan* como objetivo, destacam-se o escalonamento baseado em regra de prioridade (Kolisch e Hempel, 1996) e abordagens meta-heurísticas (procedimentos que exploram o conhecimento adquirido com a avaliação de soluções previamente visitadas), como algoritmos genéticos (Hartmann, 1998).

Abordagens de solução exata do RCPSP têm sido usados mais freqüentemente para geração de soluções *benchmark*, em face da natural dificuldade de cálculo da solução ótima do problema. A estratégia mais utilizada e objetivo são esquemas deste estudo são esquemas de enumeração implícita com *branch & bound* (Demeulemeester, 1996).

A estratégia *branch and bound* pode ser resumida da seguinte maneira: suponha o problema de sequênciar *n* tarefas independentes em um processador, dispondo-se de uma função com a qual se possa calcular o valor numérico da solução, i.e, a qualidade do escalonamento. O método *branch and bound* (B&B) inicia por gerar uma árvore de decisão com *n* ramos, representando as *n* opções possíveis para a primeira tarefa a ser executada, compondo assim o primeiro nível. Ato contínuo, cada ramo do primeiro nível se ramifica (*branch*) também, nesse caso *n-1* vezes, construindo o segundo nível, para cobrir todas as possíveis alternativas das *n-1* tarefas remanescentes como a segunda a ser processada. Prosseguindo-se dessa forma, sucessivamente até o *n*-ésimo nível, a árvore atinge *n!* ramos, um número que pode ser gigantesco a depender do número de tarefas. Em vez, portanto, de avaliar todas as soluções, o procedimento identifica e suprime (poda) regiões da árvore nas quais se pode provar não haver solução ótima e, assim, diminui o espaço de enumeração. Essa fase está associada à operação de fixação de limites (*bounding*), ao envolver o cálculo de limitantes inferiores/superiores para a solução ótima em cada um dos nós gerados na fase anterior (*branching*). Sucessivamente analisando e podando partes da árvore, o método encontra a solução exata, caso não se esgote o tempo de computação pré-estabelecido.

3. Abordagem proposta

Como alternativa para o problema de encontrar a solução ótima para o RCPS, sugerimos a utilização de uma abordagem *branch and bound* paralela, utilizando uma arquitetura de processamento colaborativa.

A vantagem em se utilizar uma arquitetura como essa é que podemos contar com uma máquina paralela "virtual" de baixo custo, fácil configuração e escalável. Contudo, a baixa velocidade da rede que interliga os nós, se comparado a máquinas paralelas, obriga a tomar alguns cuidados na construção. O principal deles é a minimização da comunicação entre os nós. Neste ponto, a abordagem *branch and bound* é perfeita, pois permite uma grande independência entre os nós de processamento.

Representando a solução do RCPSP como uma árvore de busca onde cada nó da árvore representa um escalonamento válido de atividades em um dado instante *t*, temos que a solução ótima é a menor distância entre a raiz e uma das folhas.

Cada nó computacional executa os seguintes passos: (1) encontrar todos os escalonamentos possíveis no conjunto de sucessores da atividade *i*; (2) para cada escalonamento válido s_{ij} solicitar

que um nó computacional calcule o menor *makespan* ds_{ij} até a última atividade; (3) se não houver mais atividades para escalonar, retorna a duração da atividade *i*, senão retorna $d_i + min\ \{ds_{ij}\}$.

Como ilustração do funcionamento, tomemos a rede de atividades da Figura 1. Dado o problema de calcular o menor *makespan* a partir da atividade 1, teríamos os possíveis escalonamentos: s_{11}= {2}; s_{12}= {3}; s_{13}= {4}; s_{14}= {2,3}; s_{15}= {2,4}. O escalonamento das tarefas {3, 4} simultâneamente não é válido pois fere a restrição de recursos. Assim, para cada s_{ji} é novamente calculado o menor *makespan* como se o projeto se iniciasse em s_{ji}, até que a última atividade seja alcançada.

Cada conjunto s_{ji} é representado por um nó na árvore de busca da solução. No momento em que os valores ds_{ji} do *makespan* forem encontrados para todos os conjuntos s_{ji}, o nó computacional que iniciou o cálculo para a atividade *j* seleciona o menor e soma a d_j encontrando a menor duração para o projeto.

3.1. Particionamento do problema

O particionamento do problema é feito pelos ramos da árvore. Como não há comunicação horizontal, os nós de processamento não precisam trocar mensagens entre si durante a busca por uma folha. Assim, após a ramificação (*branch*) de uma atividade, podemos ter 5 execuções em paralelo, uma para cada conjunto s_{ij} do exemplo da Figura 1, caso tivessemos 5 nós computacionais (*workers*) disponíveis. Neste caso, cada um receberia um ou mais nós do nível seguinte da árvore e continuaria a exploração a partir daí, em paralelo.

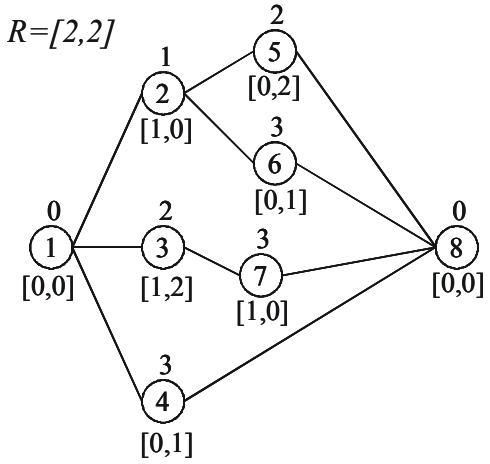

Figura 1 - Rede de atividades de um projeto

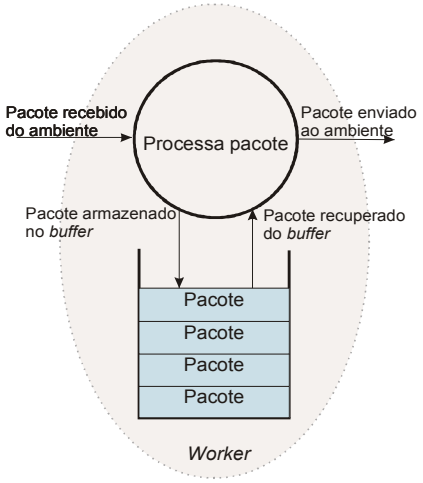

Figura 2 - Esquema de um *worker*

3.2. Master-worker hierárquico

De modo a aumentar o paralelismo e evitar que apenas um nó seja responsável pela geração de trabalho (*master*) optamos pela utilização de um modelo *master-worker* hierárquico para a alocação e geração de pacotes de trabalho. Neste modelo cada nó da rede pode, ao mesmo tempo, atuar como *master* e como *worker*. Assim, um *worker* que recebeu um nó do segundo nível deve responder ao seu *master* qual o menor escalonamento possível a partir daquele nó. Como este nó também se ramifica em vários outros, o *worker* pode requisitar que outros *workers* da rede ajudem na solução do sub-problema. O *worker* que requisitou a ajuda passa então a ser *master* para o trabalho disparado para os outros *workers*.

Cada *worker* tem a tarefa de receber um nó da árvore, expandi-lo e decidir entre repassar os nós gerados para outros *workers* ou continuar processando os nós gerados. Cada *worker* conhece o trabalho que foi realizado da raiz até o nó sendo processado. Ao chegar em uma folha, o *worker* calcula o tempo total do projeto (distância da raiz até a folha) e retorna o resultado para seu

imediato *master*. O *master* deve coletar todos os dados de seus *workers* e repassar ao seu *master* imediado o menor resultado encontrado. Este processo se repete até ao processo *master* que expandiu o nó raiz. Sua resposta será o melhor escalonamento possível de modo a minimizar a duração total do projeto.

4. Implementação

A implementação deste algoritmo em um ambiente de computação colaborativa requer a solução de dois problemas auxiliares.

O primeiro é sobre o tamanho da unidade de trabalho entregue aos *workers*. Quando se usa um meio de transmissão lento, se comparado à velocidade da CPU, como a Internet, o pacote de trabalho deve apresentar, pelo menos, uma razão de 1,5 entre o tempo gasto no trabalho computacional requerido versus a quantidade de tempo gasto para distribuir dado e código para suportar o processamento (Shread, 2002).

Pela natureza do problema, uma unidade de trabalho requer uma razoável quantidade de dado a ser enviada: tempo atual do escalonamento, atividades já executadas, atividades não executadas, atividades em execução e em que tempo iniciou, além das relações de precedência e os recursos disponíveis. O processamento realizado por um *worker* não é complexo: dada uma configuração, o *worker* deve calcular todas as opções de escalonamento possíveis para o próximo período de tempo.

Para aumentar a unidade de trabalho, cada *worker* possui um *buffer* interno que acumula diversas unidades de trabalho para si. Dessa forma, as unidades de trabalho geradas são armazenas no buffer. Sempre que o processamento de um pacote termina, o *worker* lê um novo pacote do *buffer*. Quando o *buffer* está cheio, o pacote de trabalho é enviado ao ambiente colaborativo para ser resolvido por outro *worker*. Quando não há mais pacotes no *buffer*, o *worker* verifica se há algum pacote de trabalho no ambiente para ser processado (Figura 2). Desta forma, cada pacote que um *worker* recebe gera uma grande quantidade de pacotes internos, dimunuindo o número de pacotes de trabalho na rede e reduzindo também o número de vezes que cada *worker* precisa recorrer ao ambiente em busca de mais trabalho.

Outro cuidado importante é a utilização de memória. Devido a quantidade combinatorial do problema, mesmo problemas pequenos (p.ex. 15 atividades) podem gerar uma gigantesca quantidade de nós em uma árvore de busca, mesmo utilizando um bom critério de poda. Uma solução é liberar a memória utilizada a medida que os resultados de seus filhos vão sendo encontrados. A busca em profundidade nos dá uma menor utilização de memória enquanto que a busca em largura fornece uma maior possibilidade de paralelismo. Para nos beneficiar das vantagens de cada um métodos, utilizamos uma busca híbrida.

Para cada configuração de escalonamento são geradas todas as opções possíveis de alocação para o período de tempo seguinte. Essas opções formam os pacotes de trabalho, que são inseridos no *buffer* – se estiver cheio, são disponibilizados no ambiente colaborativo. O *buffer*, na verdade, é uma pilha: sempre será consumido o último pacote que foi inserido. Dessa formar, a busca em largura é utilizada para prover o paralelismo necessário e a busca em profundidade é aplicada por cada *worker* objetivando alcançar mais rapidamente um nó folha da árvore de solução.

5. Experimento

Para a realização de testes do algoritmo, foram utilizados 12 computadores (nós computacionais) Athlon 1.1GHz com 256Mb de RAM, interligados por uma rede de 10Mbps, com sistema operacional Windows 2000. Para o experimento foi gerado um conjunto de 12 projetos RCPS através do gerador de benchmarks ProGen (Kolisch, 1995). Os testes foram executados sobre uma arquitetura de trabalho colaborativo em desenvolvemento em nossa Universidade. Esta arquitetura permite a construção de aplicações que necessitem buscar recursos computacionais disponíveis na rede, tornando transparente para a aplicação os endereços e a quantidades de nós

participantes do trabalho computacional. A arquitura é responsável por encontrar nós computacionais para resolver o problema e distribuir os pacotes de trabalho, utilizando um comunicação baseada em XML. O balanceamento de carga é feito pela arquitetura colaborativa em função da fila de pacotes que cada nó tem para processar (Pereira, 2004).

Na primeira etapa do experimento, alteramos o valor do espalhamento inicial do algoritmo, utilizando os 12 nós computacionais disponíveis. Definimos o espalhamento inicial como o número de nós na árvore de solução (pacotes de trabalho) que serão explorados em largura, antes de ser utilizado o *buffer* interno (busca em profundidade). A segunda etapa do experimento consistiu em medir, para cada projeto do lote escolhido, o tempo de termino do problema e o tempo gasto por cada nó computacional. O experimento foi realizado variando o número de nós computacionais de 2 a 12.

6. Resultados

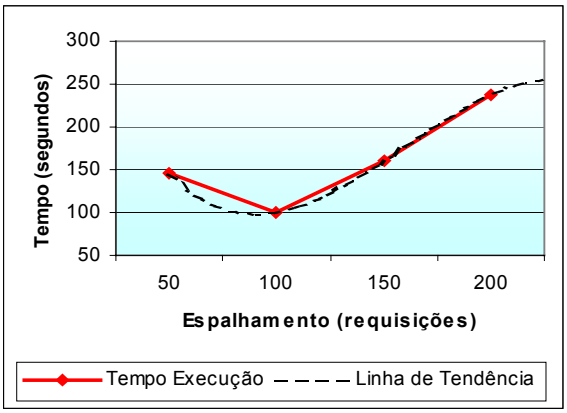

Figura 3 - Variação da duração total do problema em função do espalhamento inicial

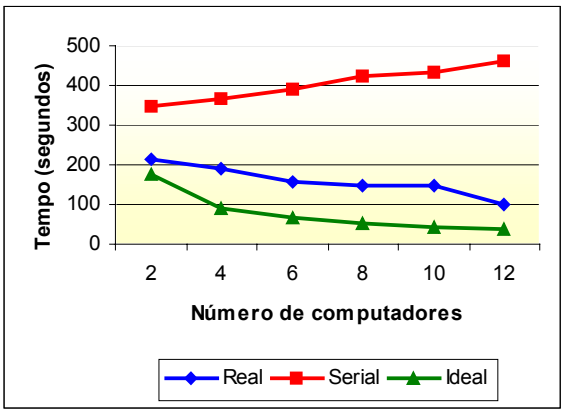

Figura 4 - Variação da duração do problema em função do número de computadores

7. Discussão

A Figura 3 mostra a variação do tempo médio gasto para a solução ótima do lote de teste em função do espalhamento inicial. Com o aumento do espalhamento inicial, aumentamos a quantidade de pacotes iniciais geradas, favorecendo assim o paralelismo. Contudo, mantendo o número de nós constante, há um momento em que não há nós ociosos que possam resolver pacotes de trabalho. A quantidade excedentende de pacotes, além de não poderem ser tratadas em paralelo, inserem um gasto a mais de processamento, no algoritmo e na arquitetura colaborativa, devido a geração e gerenciamento das requisições, fazendo com que o tempo de execução médio aumente, conforme mostra a Figura 3. O tempo de execução real é o tempo decorrido entre o disparo da primeira requisição até a obtenção do cronograma ótimo para o problema.

Na segunda etapa do experimento fixamos o espalhamento em 100 requisições, por ser o melhor parâmetro para o lote de problemas utilizando 12 nós computacionais, conforme resultado da etapa anterior do experimento. No gráfico da Figura 4 podemos comparar o comportamento do tempo de execução do algortimo versus o tempo serial e o tempo ideal. Definimos tempo serial como sendo a soma da quantidade de tempo que cada nó computacional gasta resolvendo o problema, não levando em consideração o tempo de envio do pacote pela rede. Tempo ideal é o tempo serial dividido pelo número de nós e funciona como limite inferior para o tempo de execução. A variação do tempo serial em função do número de nós demonstra o peso do gasto de gerênciamento de requisições, pois quanto maior o número de nós na rede, maior a quantidade de pacotes gerados, lembrando que no modelo adotado, todo nó funciona como *master* e *worker*. A

obtenção de tempos muito próximos com 8 e 10 nós computacionais deve-se ao fato que, para os parâmetros utilizados (lote de projetos e espalhamento inicial) o aumento no paralelismo não compensa o aumento no gasto com requisições. Ainda, o ganho alcançado com 12 nós é devido ao parâmetro de espalhamento, já que utilizamos o valor que apresentou melhor resultado para a configuração de 12-nós.

As características da árvore de busca do problema influência diretamente na performance do algoritmo. O paralelismo está diretamente ligado a largura da árvore. Uma árvore que possui poucos nós irmãos terá um fator menor de paralelismo, tendendo para uma execução serial. A altura da árvore de busca também influência a performance. Uma árvore com poucos níveis tende a ser resolvida muito rapidamente, não compensando os gastos com geração e envio de pacotes de trabalho. O algoritmo irá apresentar melhor performance para grandes problemas que possuam árvores de busca largas – favorecendo o paralelismo – e profundas – gerando uma quantidade de trabalho que não é trivialmente resolvida pelos nós computacionais.

8. Conclusão

O experimento mostrou a escalabilidade do algoritmo, onde o aumento do número de nós levou a uma redução do tempo total. A partir destes resultados, podemos razoavelmente inferir que a inserção de novos nós irá reduzir o tempo total requerido para resolver problemas de maior porte.

O experimento mostrou também a necessidade de melhorias no algoritmo, de forma a melhorar o desempenho com o aumento do número de nós. Estamos trabalhando em um novo algoritmo adaptativo que usa informação sobre a carga de trabalho dos workers para melhorar a alocação dos pacotes de trabalho.

Finalmente, o podemos observar que a aplicação deste método não se restringe ao RCPSP. As técnicas utilizadas tais como: a criação de um buffer interno a a utilização do modelo master-worker hierárquico podem ser aplicadas para resolver qualquer problema que utilize a abordagem B&B.

9. Referências

Fleszar, K., Hindi K. (2002). Solving the resource-constrained project-scheduling problem by a variable neighborhood search. European Journal of Operational Research.

Herroelen, W.S., *et. al.* (1997). Project network models with disconted cash flows – A guide tour through recent developments. European Journal of Operational Research, 100:97-121.

Demeulemeester, E., *et. al.* (1999). A classification scheme for project scheduling, in J. Węglarz (Ed.), Project Scheduling: Recent models, algorithms and applications, Kluwer, Dordrecht.

Garey, M., Johnson, D. (1979). Computer and intractability: A guide to the theory of NP-completeness. W.H. Freeman, San Francisco.

Sarmenta, L. F. G. (2001). Volunteer Computing. Tese de doutorado Massachusetts Institute of Technology.

Shread, P. (2002). What Jobs Are Right For Internet Grid Computing Services?. Grid Computing Planet News, January 18. http://gridcomputingplanet.com/news/article.php/958281

Kolisch, R., Hempel, K. (1996). Finite scheduling capabilities of commercial project management systems, manuskripte aus den Instituten für Betriebswirtschaftslehre Universität Kiel, 397.

Hartmann, S. (1998). A competitive genetic algorithm for resource-constrained project scheduling, Naval Research Logistics, 45: 733-750.

Demeulemeester, E., *et. al.* (1996). Optimal procedures for the discrete time/cost trade-off problem in project networks, European Journal of Operational Recearch, 88: 50-68.

Kolisch, R., *et. al.* (1995). Characterization and generation of a general class of resource-constrained project scheduling problems. Management Science 41, 1693±1703.

Pereira, F. C. (2004). IeC – Uma arquitetura XML para computação voluntária P2P, Tese de Mestrado, a ser publicada.