IMPLEMENTAÇÃO DE UM AMBIENTE DE EXECUÇÃO MULTITHREAD PARA APLICAÇÕES COM PADRÕES IRREGULARES DE PARALELISMO

CÍCERO CAMARGO; ALAN ARAÚJO; GERSON CAVALHEIRO
Universidade Federal de Pelotas – {cadscamargo,asdaraujo,gerson.cavalheiro}@inf.ufpel.edu.br

1. INTRODUÇÃO

Este trabalho aborda a implementação de ambientes multithread que utilizam de estratégias de escalonamento de lista como mecanismo de exploração de programas paralelos em arquiteturas *multicore*. Ferramentas como Intel® Cilk Plus (INTEL, 2012b), OpenMP (CHANDRA et al., 2001) e Intel® Threading Building Blocks (TBB) (INTEL, 2012a) são exemplos de ferramentas que fornecem interfaces de programação multithread adequadas para a descrição da concorrência presente em uma aplicação e ambientes de execução dotados deste tipo de recurso de escalonamento. Cilk Plus e OpenMP são ferramentas que impõem uma estrutura de programa paralelo na forma de *fork-joins* aninhados e implementam o ambiente de execução otimizado para esta estrutura de programa. TBB, por outro lado, oferece uma interface de programação não restritiva, que permite ao programador descrever aplicações com vários padrões de paralelismo, fornecendo um ambiente de execução capaz de escalonar a aplicação de maneira eficiente sobre o hardware paralelo.

Algoritmos de lista (GRAHAM, 1966) possuem eficiência comprovada no escalonamento de programas descritos em grafos de fluxo de dados sobre arquiteturas multiprocessadas. Cilk Plus, OpenMP e TBB são exemplos de ferramentas comerciais em evidência, as quais empregam com sucesso esta técnica em ambientes multithread dinâmicos. De maneira similar, Anahy (CAVALHEIRO et al., 2007) é uma ferramenta desenvolvida em meio acadêmico, que também utiliza técnicas de algoritmos de lista no escalonamento de threads em tempo de execução. Sua interface de programação é baseada somente em operações *fork* e *join*, as quais permitem a criação/sincronização de múltiplos threads em uma única operação. Em Anahy pode-se descrever aplicações com padrões irregulares de paralelismo, pois um thread não precisa ser sincronizado estritamente pelo thread que o criou, nem tampouco por um único thread.

Os ambientes multithread citados são focados em extrair alto desempenho na execução de programas concorrentes sobre arquiteturas paralelas. Mais do que isso, assumem que o programador descreve a concorrência da aplicação sem considerar a arquitetura que a executará. Tal abordagem geralmente resulta em aplicações onde os threads são criados em tempo de execução, em grande quantidade e fina granularidade, isto é, para realizar pouca carga computacional. Assim, tanto as estruturas de dados quanto as rotinas do ambiente de execução devem ser otimizadas ao máximo, para inserir a menor quantidade de sobrecustos nas tarefas de escalonamento e gerenciamento da concorrência.

Neste trabalho serão discutidas algumas alternativas de projeto para a implementação de um ambiente de execução multithread e o impacto destas alternativas no sobrecusto gerado pelas rotinas do ambiente.

2. MATERIAL E MÉTODOS

O modelo *multithread* que consideraremos assume que o programa inicia e termina com a execução de um único thread, representado pela função principal de um programa. A partir deste thread principal, novos threads podem ser criados

ou sincronizados pelo uso de primitivas *fork* e *join*, respectivamente. Um grafo de threads, que é dinamicamente construído a cada chamada a *fork* e *join*, é mantido pelo escalonador para que este possa controlar a execução das atividades concorrentes do programa. Este grafo também é usado na escolha de um novo thread a ser executado quando um processador fica ocioso. A escolha deste thread deve ser feita de maneira rápida, de modo a não inserir sobrecustos na execução do programa, e consciente, para que não haja atrasos na execução em função da má seleção do thread a ser executado.

Como alternativas de implementação, o grafo de execução do programa pode ser mantido como uma entidade centralizada no ambiente, ou pode ser distribuído localmente entre os processadores, cada um mantendo uma porção local do grafo. O grafo centralizado pode ser, ainda, mantido por um escalonador, reativo a eventos de escalonamento disparado pelos processadores quando das criações, das sincronizações e dos términos de threads, ou pode ser de acesso compartilhado por todos os processadores, os quais agem proativamente nos mesmos pontos de escalonamento citados. As distribuições do grafo e do mecanismo de escalonamento no ambiente influem diretamente na maneira como as primitivas *fork* e *join* devem ser implementadas e nos seus custos operacionais. Estas diferentes alternativas são o objeto de estudo deste trabalho.

2.1 Fork: criação de um novo thread

Quando a primitiva *fork* é invocada, um novo descritor de thread deve ser inserido no grafo e uma operação de escalonamento pode ser iniciada, caso exista pelo menos um processador ocioso. O processador que cria o novo thread pode escolher por continuar a execução do thread que invocou a primitiva *fork*, sendo o novo thread inserido em um repositório de threads prontos (estratégia *help-first*), ou começar a execução do novo thread e "anunciar" que o antigo thread está apto a ser continuado (estratégia *work-first*). Caso o grafo seja mantido como uma estrutura de acesso global aos processadores, estes necessitam adquirir um *lock* para realizar a inserção do novo thread no grafo. Da mesma forma, se o grafo é mantido por apenas uma entidade no ambiente, por exemplo o escalonador, precisa haver uma sincronização entre o processador e o escalonador para que o segundo receba do primeiro o thread a ser inserido no grafo. Ambas as opções geram sobrecustos de sincronização.

Quando o grafo se encontra distribuído em estruturas locais aos processadores a inserção de um novo thread pode ser mais eficiente. Cilk, por exemplo, mantém estruturas locais aos processadores nas quais os novos threads são inseridos. Na maioria das vezes não é necessária a obtenção de *locks* para tal operação, o que resulta em grande eficiência, porém alguns cuidados são tomados na implementação para prevenir que a consistência das estruturas locais de um processador não seja invalidada por outro processador tentando roubar trabalho.

2.2 Join: sincronização entre threads

No momento da execução de uma primitiva *join* duas situações podem ocorrer: ou o thread que recebeu o *join* já terminou sua execução e o processador que executou a primitiva pode continuar a execução do thread atual, ou o thread não está pronto porque sua execução não foi começada ou está em andamento.

A primeira situação não gera sobrecustos de execução, uma vez que a memória é compartilhada e não importa onde o thread que recebe o *join* está armazenado (em uma estrutura global ou local aos processadores), desde que o

processador que executa o *join* consiga conferir o estado do thread que sofre o *join* de maneira atômica.

A segunda situação gera sobrecustos de escalonamento, pois o processador que executa o *join* precisa interromper o thread atual, o qual depende de resultados ainda não produzidos pelo thread que sofreu o *join*. Como o processador torna-se ocioso, é iniciada a busca de um novo thread para que este continue, possivelmente focando em threads que contribuam para o término do thread que sofreu o *join* e fez com que o processador atual ficasse ocioso.

Outra questão importante é a remoção de um thread do grafo. Em ambientes restritos a computações com *fork-joins* aninhados a remoção pode ser efetuada tão logo o *join* tenha sido executado com sucesso. Em ambientes como Anahy ou TBB, onde o grafo pode ser bastante irregular e um thread pode sofrer *join* a partir de diferentes threads, é preciso que o programador estabeleça a quantidade de *joins* que um thread sofrerá, e cada *join* executado decremente este valor atomicamente. Assim, o ambiente pode garantir que os resultados produzidos por um thread já foram lidos corretamente nos pontos certos do programa.

2.3 A busca por trabalho

Quando um processador fica ocioso, este busca por threads que estejam prontos para executar. Um thread está pronto para executar quando não possui dependências insatisfeitas, o que é verdade em duas situações: *(i)* o thread acabou de ser criado; *(ii)* o thread foi parcialmente executado, bloqueou-se ao executar um *join* sobre um segundo thread T que não havia terminado, e, logo após, T terminou. Desconsiderando a situação inicial do programa, os processadores ficam ociosos sempre que terminam a execução de um thread ou executam *join* sobre um thread não terminado. Estas diferentes situações podem influenciar na maneira com que o processador que ficou ocioso buscará mais trabalho.

A estratégia de escalonamento diz respeito a como a busca por trabalho será realizada. Esta estratégia pode estar implementada de forma reativa, onde os processadores avisam o escalonador, uma entidade à parte, de que necessitam de trabalho e aguardam até que este lhes retorne alguma resposta. A estratégia pode ser proativa, onde os processadores farão a busca do próprio trabalho. Em ambos os casos, técnicas diferentes devem ser utilizadas quando o grafo é centralizado ou distribuído pelas estruturas de dados locais a cada processador. No primeiro caso, geralmente, observa-se mais contenção no ambiente de execução, uma vez que cada processador deve obter acesso em exclusão mútua ao grafo, para fazer uma busca relativamente complexa.

2.4 O término de um thread

Quando um processador termina a execução de um thread *t* uma série de eventos devem ocorrer. Primeiro, os eventuais threads que ficaram bloqueados ao tentar executar *join* sobre *t* devem voltar ao *pool* de threads prontos para execução. Segundo, o processador deve notificar ao escalonador que está ocioso devido ao término do thread *t* para que este possa buscar trabalho no grafo. Entre o primeiro e o segundo passo, podemos inserir uma etapa de otimização para o seguinte cenário: no thread *s* é executado um *join* sobre *t*, o qual está pronto para execução; neste caso o processador ignora o escalonador e executa *t*, retomando a execução de *s* imediatamente após o término de *t*. Apesar de eliminar sobrecustos, esta otimização deve ser considerada com cuidado, pois pode interferir de maneira negativa no escalonamento global do programa.

3. RESULTADOS E DISCUSSÕES

Athreads (CAVALHEIRO; VIÇOSA, 2007) é uma implementação do ambiente Anahy para máquinas *multicore*, a qual utiliza uma interface de programação baseada no padrão POSIX (IEEE 1003) para threads. Embora sua interface seja bastante conveniente, possibilitando até a tradução automática de alguns programas POSIX para programas Athreads, seu ambiente de execução agrega diversos recursos *ad-hoc*, os quais geram sobrecustos que inviabilizam a execução de programas com paralelismo de granularidade muito fina.

Uma nova implementação do ambiente Anahy se encontra em desenvolvimento. A implementação contempla técnicas apresentadas na Seção 2 visando associar uma arquitetura de software com o grafo distribuído entre os processadores e o suporte de escalonamento provido por algoritmos de lista.

Testes preliminares indicaram que um ambiente com processadores proativos na busca de trabalho resulta em menos contenção, porém dificulta o tratamento de situações como a de não haver trabalho e o processador ter de se desativar para poupar energia, por exemplo.

4. CONCLUSÕES

Este trabalho apresentou diversos conceitos relativos à implementação de um ambiente de execução *multithread* para arquiteturas com memória compartilhada. Diversas combinações das estratégias são possíveis e estão sendo avaliadas nesta nova implementação.

Uma vez que a implementação do ambiente esteja finalizada e validada, será possível testar o impacto de diversas estratégias de escalonamento associadas aos modos de operação estabelecidos para o ambiente. Outros fatores como a otimização do espaço de memória utilizado pelas estruturas de dados do ambiente podem ser analisados futuramente.

5. REFERÊNCIAS BIBLIOGRÁFICAS

CAVALHEIRO, G. G. H.; GASPARY, L. P.; CARDOZO, M. A.; CORDEIRO, O. C. Anahy: a programming environment for cluster computing. Em: VII HPCS, Berlin. **Anais do...**, 2007.

CAVALHEIRO, G. G. H.; VIÇOSA, E. Athreads: a dataflow programming interface for multiprocessors. In: LTPD 2007, Gramado-RS. **Anais do...** [S.l.: s.n.], 2007.

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; MCDONALD, J.; MENON, R. **Parallel Programming in OpenMP**. San Francisco: Morgan Kaufmann, 2001.

GRAHAM, R. Bounds on the performance of scheduling algorithms. Computer and Job-Shop Scheduling Theory, [S.l.], p.165–227, 1976.

INTEL, C. **Intel(R) Threading Building Blocks**: reference manual. Acessado em Janeiro/2012, http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf.

INTEL, C. **Using Intel(R) Cilk(TM) Plus**. Acessado em Janeiro/2012, http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/mac/cref_cls/common/cilk_bk_using_cilk.htm.