Implementação paralela de algoritmo de busca

João Marcelo Ceron*
Ney Lemke
Gerson Geraldo H. Cavalheiro

Programa de Pós-Graduação em Computação Aplicada – PIPCA
Centro de Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
São Leopoldo – RS – Brasil
{jmarcelo,lemke,gersonc}@exatas.unisinos.br

Introdução

Embora o rápido progresso tecnológico disponibiliza microprocessadores cada vez mais velozes, a capacidade de processamento seqüencial ainda é limitada para muitas aplicações. Soluções a estes problemas são buscadas em implementações sobre arquiteturas paralelas. Buscam-se implementações que permitam obter índices de desempenho satisfatórios. Para isso deve-se explorar ao extremo a concorrência das aplicações em ambientes multiprogramados, que incrementam ao máximo a utilização dos recursos de processamento.

Um dos objetivos da programação concorrente é o ganho de desempenho, permitindo distribuir partes do programa em execução sobre os recursos da arquitetura [SPR 97]. Neste contexto, o paradigma de programação concorrente permite descrever aplicações sobre a forma de processos concorrentes, capazes de serem utilizadas sobre arquiteturas paralelas.

Diversas ferramentas possibilitam a construção de programas concorrentes [CAV 01], entre elas, a linguagem Java que disponibiliza o recurso de multiprogramação leve (*threads*) [O'RE 97].

Neste trabalho é descrita uma implementação concorrente para uma varredura numa região bi-dimensional do espaço [FER 01]. Ela pode ser representada pela existência de um grupo de bombeiros procurando focos de incêndio, em um terreno delimitado. O restante deste trabalho está organizado na apresentação do algoritmo, implementação concorrente e alguns resultados de desempenho obtidos.

O algoritmo do bombeiro

Este algoritmo é a solução para o seguinte problema: uma determinada quantidade de bombeiros encontra-se em um terreno, onde existem focos de incêndio; e cada foco de incêndio cobre de fumaça uma determinada área ao redor. O objetivo é fazer com que os bombeiros apaguem o mais rápido possível, todos os focos de incêndio.

*BIC-FAPERGS

As entidades empregadas na implementação concorrente deste algoritmo são descritas a seguir. A representação gráfica do problema encontra-se na figura 1.

O bombeiro: é a entidade ativa do problema. Ele percorre um terreno em busca de focos de incêndio, podendo mover-se em todas as direções, uma célula por vez. Ao encontrar um foco, ele o apaga, retirando-o do terreno (assim como a fumaça liberada pelo mesmo existente nas suas proximidades).

O terreno: é bi-dimensional, de dimensão m x n, composto por células, todas estas compartilhadas. Cada uma pode ser ocupada por apenas um bombeiro em um determinado instante de tempo. O acesso a cada célula implica na execução de código em uma zona crítica por manipular dados compartilhados entre todos os bombeiros. O acesso a uma célula deve, portanto, ser sincronizada, evitando inconsistência de dados. Além do bombeiro, ela poderá conter somente fumaça, foco de incêndio ou ainda estar vaga.

Focos de Incêndio: estão dispersos no terreno de forma aleatória. Assume-se que não possam existir dois (ou mais) focos em uma célula, o que não impede focos em células vizinhas. Eles exalam uma fumaça, a qual pode ser sentida pelos bombeiros nas proximidades do incêndio. Caso a fumaça de dois focos se sobreponham em uma célula, esta ”quantidade maior de fumaça ” pode ser notada, e deve ser considerada.

A informação fumaça também é utilizada, para auxiliar o bombeiro na sua busca. Sendo que ele possui um certo nível de inteligência, a qual o faz, logo que sentir a fumaça, procurar o foco de incêndio que deve se encontrar próximo.

A inteligência do bombeiro: foi implementada em dois níveis. No nível individual, uma forma de tomar a decisão sobre o que fazer, e no nível de cooperador, interagindo com a equipe. A inteligência no nível individual, faz com que ele tome decisões assim que entra em contato com a fumaça, guiando-o até o foco que se encontra mais próximo a ele, exterminando-o. No nível cooperador, ele interage com a equipe, sabendo-se que os focos não vão ser ampliados, e que sempre que um bombeiro passar conseguirá apagá-lo. Cada indivíduo deixa um rastro por onde passa, fazendo que seus companheiros não passem a procurar focos de incêndio por onde outro já passou. O critério para que a equipe pare de procurar por focos, através de uma informação compartilhada entre todos, é que todos os focos de incêndio já tenham sido exterminados.

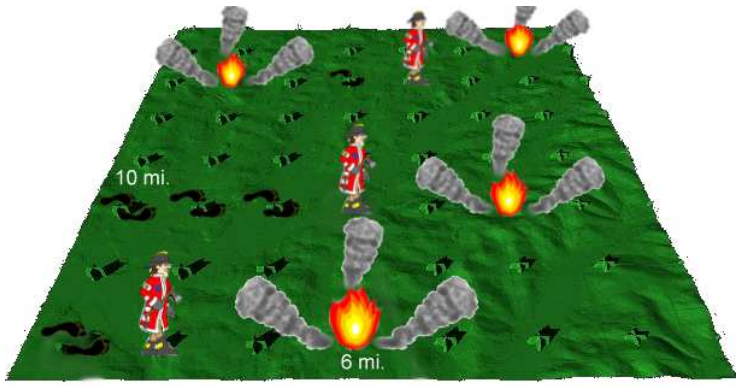

Figura 1: Ambiente do algoritmo do bombeiro

Implementação Concorrente

Desde 1995 no seu surgimento, Java, está adquirindo um crescente número de programadores. Um das razões, para isso é a facilidade da sua programação concorrente, tornando-se uma opção para o processamento de alto desempenho (PAD). Outro fator importante é sua portabilidade, que é atribuída ao seu interpretador. Java permite que suas *threads* sejam criadas de diferentes formas. A forma utilizada foi estender a classe bombeiro a classe Thread (nativa em Java), nesta classe utiliza-se o método start(), o qual faz com que a *thread* inicialize o que está contido no método run(). É importante salientar o método join (), que é responsável pela espera do fim da execução de todas as *threads* iniciadas. Quando muitas *threads* estão concorrendo pelo acesso de um determinado recurso, poderá ocorrer, devido a ordem de como que elas forem escalonadas pelo escalonador do sistema, inconsistência de dados.

Este problema poderá ser solucionado com a sincronização das *threads*. A sincronização em Java é suportada por um esquema próximo a um monitor; para um objeto são determinados quais métodos não podem ser executados simultaneamente por threads distintas. Isto é feito para garantir a correta manipulação dos atributos dos objetos (sessões críticas). No algoritmo do bombeiro, foi implementada um matriz bi-dimensional da classe terreno, a qual representava a situação de cada célula da matriz, com a sua situação, podendo estar livre, ocupada por um bombeiro, conter foco de incêndio ou conter fumaça.

Na classe matriz, são implementados métodos que permitem o compartilhamento de informações entre os bombeiros. Os dados compartilhados foram o número total de focos de incêndio, assim como a manipulação dos mesmos, e a verificação da posição dos bombeiros. Esta classe também é responsável pela implementação da geração aleatória de focos de incêndio, com o total previamente definido, também pela colocação dos respectivos sinais de fumaça. Outra função é a criação de um vetor de *threads* e a invocação das mesmas.

Outra classe implementada é a classe bombeiro, que é um fluxo de execução criado na classe matriz. A classe bombeiro é responsável por percorrer toda a tabela, começando de uma posição aleatória, até que a quantidade de focos seja totalmente extinta. O bombeiro poderá interagir com o terreno e perceber se há sinais de fumaça. Localizando estes sinais, o bombeiro guardará a sua posição, irá vasculhar ao redor da fumaça a procura do foco, localizado irá apagá-lo e retirar o sinal da fumaça. Outro ponto incrementado foi o rastro do bombeiro, por onde um bombeiro passa é deixado um rastro, fazendo com que os outros bombeiros não procurem por focos em posições já visitadas. Na figura 2, são apresentados os resultados obtidos.

Em trabalhos futuros, novos algoritmos implementando a inteligência do bombeiro podem vir a ser implementados. Outro problema a ser tratado diz respeito ao consumo da memória, explorando apenas o uso das *threads*, a complexidade da execução fica limitada à quantidade de memória de um nodo. Uma nova implementação em agregados de computadores permitirá a simulação de processos de busca em terrenos ainda maiores. Os testes foram feitos em uma máquina Linux, bi-processada de 1 Ghz. No caso estudado, foi implementado uma matriz com dimensões 50 por 50, com 200 focos de incêndio. A complexidade do problema não foi alterada, somente o número de bombeiros em busca

por focos. Devido à aleatoriedade dos dados, foi calculada a média aritmética dos resultados, somente após plotados no gráfico. Como pode ser visto no gráfico, podemos notar um ganho de 264 % utilizando três threads.

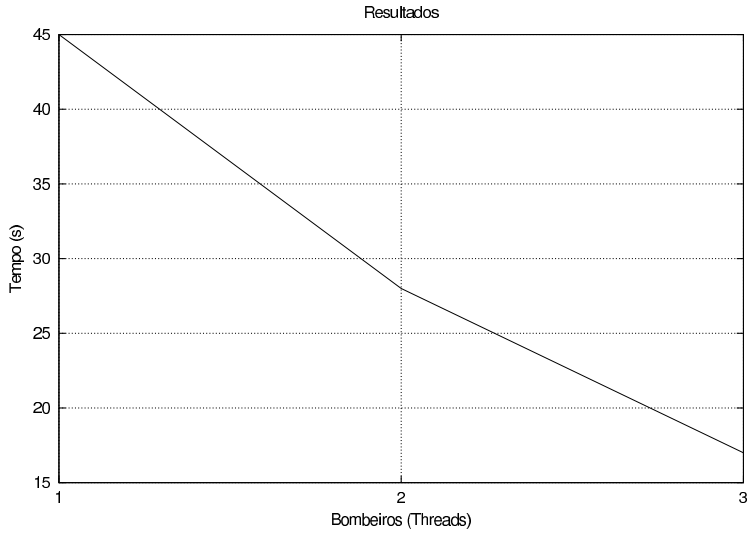

Figura 2: Resultados

Conclusão

Muitos sistemas operacionais modernos fornecem ferramentas para que um processo possa conter múltiplas *threads* de controle. Neste trabalho foi possível observar, através dos resultados obtidos, que aumentando a multiprogramação se pode reduzir consideravelmente o tempo de execução de uma tarefa, fazendo o uso das *threads*. Com o aumento da complexidade dos problemas, e seu elevado tempo computacional, tornando-se crítico, uma proposta que este trabalho abordou foi a utilização de *threads*, pela sua simples implementação e ótimos resultados obtidos.

Referências

[CAV 01] CAVALHEIRO, G. G. H. Introdução à programação paralela e distribuída. In: ERAD 01, 01, Gramado. **Anais...** [S.l.: s.n.], 01.

[FER 01] FERNANDES, K. C. **Systèmes multi-agentes hybrides**: une approche pour la conception de systèmes complexes. 01. doutorado — UFRJ, Rio de Janeiro.

[O'RE 97] O'REILLY (Ed.). **Java threads**. Sebastopol: Scott Oaks and Henry Wong, 97.

[SPR 97] SPRINGER-VERLAG (Ed.). **On concurrent programming**. New York: Fred B. Schneider, 1997.