Insper

11 - Granularidade de tarefas

SuperComputação 2018-2

Igor Montagner, Luciano Soares

Parte 1 - experimentos com diferentes tipos de arquivos

Na parte 1 do roteiro 10 fizemos um programa que busca strings em arquivos presentes em um diretório. Ao realizar a paralelização do código notamos que houve pouquíssimo (ou nenhum) ganho de desempenho. Vamos verificar o desempenho de nosso programa em duas situações:

- poucos arquivos, mas de tamanho grande
- muitos arquivos pequenos

Você pode gerar arquivos de teste para esta tarefa usando os scripts *make-big-file.py* e *make-small-files.py*. Ambos geram arquivos com linhas de até 100 caracteres contendo strings aleatórias, simulando arquivos de texto ou logs.

Nos dois exercícios abaixo gere gráficos para facilitar a compreensão do desempenho do programa e comente o resultado obtido na folha.

Exercício 1: em uma pasta separada, gere quantidades incrementalmente maiores de arquivos pequenos e analize o desempenho de seu programa.

Exercício 2: em pastas separadas, gere 20 arquivos de tamanho incrementalmente grande (desde 1Mb até centenas de megabytes). Analize o desempenho do seu programa nesta situação.

Exercício 3: em algum dos dois casos houve ganho de desempenho na paralelização? Explique sua conclusão.

Granularidade e divisão de tarefas

Diferentemente dos problemas anteriores, não houve ganho de desempenho simplesmente ao adicionar mais threads. Isto ocorre pois a busca em arquivos é fundamentalmente diferente dos problemas que tratamos anteriormente em relação a

- número de tarefas (se muito, milhares)
- complexidade de cada tarefa (variável, depende do conteúdo do arquivo)
- compartilhamento de recursos

A criação de tarefas (tanto usando `task` como `parallel for`) é cara e só é vantajosa quando

- o número de tarefas é grande
- cada tarefa é minimamente complexa

Em especial no caso do `parallel for`, o runtime do OpenMP faz a divisão de tarefas para nós, criando tarefas minimamente complexas que *agregam várias iterações do loop*. A diretiva `schedule`, que vimos nas aulas expositivas, oferece opções para modificar a maneira com que as iterações do for são distribuídas entre as threads.

No caso da leitura de arquivos, ao lançarmos uma tarefa para cada arquivo podemos estar criando muitas tarefas de baixa complexidade. Ou seja, como as tarefas acabam muito rápido a troca entre tarefas consome tempo significativo em relação ao tempo de cada tarefa. Além disto, temos diversas threads acessando o disco, um recurso que só pode ser usado uma thread por vez. A combinação de ambos fatores cria uma situação pouco favorável as técnicas de paralelização que vimos até o momento.

Chamamos de **granularidade** a quantidade de computação executada por cada tarefa. Uma divisão de tarefas com *alta* granularidade cria tarefas que fazem *pouco* trabalho, mas que podem ser paralelizadas facilmente. Um fator importante a ser considerado é o custo de lançar e alternar entre tarefas, que pode ser significativo em relação aos ganhos obtidos pela paralelização massiva.

Uma divisão de *baixa* granularidade cria tarefas que fazem *muito* trabalho. Isto dificulta o balanceamento da paralelização, mas diminui significativamente custos de troca de tarefas e comunicação. É especialmente vantajoso

- em sistemas distribuídos, em que o custo de comunicação é alto
- quando usamos um recurso compartilhado

Controlar a granularidade da divisão de tarefas é essencial para obter ganhos de desempenho significativos ao trabalhar com computação paralela.