

Resumo da Spring School in Advanced Computing TACC @ IST 2009

Ângela Canas (MARETEC), Rosa Trancoso (Secção de Ambiente e Energia – IST)

INTRODUÇÃO	1
INTRODUÇÃO AO TACC E À COMPUTAÇÃO PARALELA	2
INTRODUÇÃO À PROGRAMAÇÃO USANDO OPENMP	4
ENGENHARIA DE OPTIMIZAÇÃO E DESEMPENHO PARA APLICAÇÕES CIENTÍFICAS...	5
INTRODUÇÃO À PROGRAMAÇÃO USANDO MPI	6
DESEMPENHO DE ESCALABILIDADE	7
INTRODUÇÃO À VISUALIZAÇÃO CIENTÍFICA.....	8

Introdução

Esta workshop decorreu no Instituto Superior Técnico a 25 e 26 de Maio tendo como objectivo o estreitar da interacção entre a comunidade portuguesa que usa computação avançada e um dos mais importantes centros mundiais para Computação de Alta Performance, o Texas Advanced Computing Centre (TACC) da Universidade do Texas, Austin.

O programa compreendeu os seguintes tópicos, apresentados em sessões teóricas e na maior parte dos casos também em sessões práticas:

- introdução ao TACC e à computação paralela (teórica);
- ambiente de trabalho do cluster RANGER do TACC (prática);
- introdução à programação usando OpenMP (teórica e prática);
- engenharia de optimização e desempenho para aplicações científicas (teórica);
- introdução à programação usando MPI (teórica e prática);
- desempenho de escalabilidade (teórica e prática);
- introdução à visualização científica (teórica e prática).

De seguida apresenta-se o essencial retido dos principais tópicos. Para informação mais detalhada consultar as apresentações fornecidas em anexo.

Introdução ao TACC e à Computação Paralela

Este tópico foi apresentado por Dan Stanzione (dan@tacc.utexas.edu).

Foi feita uma pequena introdução ao TACC, que tem como objectivo fundamental permitir descobertas científicas e o desenvolvimento da sociedade através de aplicação de tecnologias avançadas de computação, através do fornecimento de recursos como máquinas para computação avançada e serviços como apoio tecnológico.

Este centro tem várias áreas de especialização: Computação de Alto Desempenho, com aplicações especialmente na área de biomedicina, modelação meteorológica/oceânica e química computacional; Análise de Dados e Informação, em que se inclui entre outros a visualização de dados científicos; Computação Distribuída e Colaborativa.

Foram apresentados os principais recursos computacionais existentes no TACC. Destes pode-se destacar o RANGER, um Sun Constellation System com 62976 *cores* e 2GB de memória por *core*, distribuídos em 3936 nós com 16 processadores cada, sendo actualmente o super computador em 4ª posição no TOP500.

O acesso aos recursos e serviços TACC (armazenamento, computação e visualização) está especialmente direccionado para instituições académicas dos Estados Unidos da América mas é possível estabelecerem-se acordos com colaboradores internacionais. Os serviços oferecidos não são concorrentes dos sistemas *cloud* da Google.

Relativamente à computação paralela referiu-se que actualmente os sistemas de Computação de Alto Desempenho são na sua quase totalidade sistemas paralelizados. A tendência é no futuro a paralelização ser total.

Consistindo a computação paralela no uso de vários processadores ou computadores para uma tarefa comum as principais vantagens estão ao nível da solução de problemas de dimensão maior (*capability*) ou da solução de mais casos ou de forma mais rápida (*capacity*). As vantagens só serão importantes para problemas que demorem algum tempo a ser processados: ex. em processos de curta duração (minutos) a paralelização (de larga escala) pode não ajudar devido aos *overheads*.

Foram referidos os limites da paralelização. Os limites teóricos consistem na **Lei de Amdahl**: o desempenho do programa paralelizado encontra-se limitado pela quantidade de secções sequenciais (não paralelizadas), sendo que estas secções estão sempre presentes na grande maioria dos programas. Se a parte paralelizada de um código for reduzida o número de processadores que é usado nessa parte não tem grande influência prática no desempenho da paralelização, não compensado usar neste caso um sistema de larga escala para o

processamento. Verifica-se que há medida que a dimensão de um problema computacional a porção paralelizada deve aumentar.

Na prática o limite da Lei de Amdahl nunca é atingido. A razão para tal reside essencialmente na existência de custos das comunicações entre processadores e da distribuição de carga/sincronização.

Os computadores paralelos podem ser classificados de várias formas. A classificação tradicional é a de Flynn, que considera o número de instruções e de dados considerados por estes computadores. Contudo, tendo em conta que a maioria das máquinas paralelas consideram múltiplas instruções e múltiplos dados, a classificação mais útil é que organiza os computadores paralelos em termos do modelo de memória usado: **memória partilhada** (cada processador não tem memória própria) ou **memória distribuída** (cada processador tem memória local). Nos sistemas distribuídos, em que se englobam a maioria dos sistemas de larga escala, todos os processadores podem funcionar à mesma velocidade.

Os dois extremos serão os SMPs (*Symmetric MultiProcessor*), com memória partilhada e pequeno número de processadores têm como vantagens uma fácil programação e reduzido custo de produção (caso dos computadores *quad-cores*), e os clusters, que têm uma programação difícil (devido à memória distribuída e à necessidade de comunicação entre processadores) mas são fáceis de construir e suportam um grande número de processadores.

Tanto os sistemas de memória partilhada como os sistemas de memória distribuída podem ser programados de modo a desempenharem operações independentes em dados diferentes (modo MIMD, paralelismo de tarefas) ou para desempenharem a mesma operação em dados diferentes (SIMD, paralelismo de dados). O modo de programação mais comum é o paralelismo de dados, que é o mais simples de programar.

O **OpenMP** é um *standard* da programação paralela em sistemas de memória partilhada. Trata-se não de uma linguagem mas apenas de um conjunto de directivas. Como a memória é partilhada a paralelização pode gerar conflitos no acesso a variáveis partilhadas. Estes conflitos têm de ser resolvidos ao nível da programação, linguagem e/ou arquitectura.

O **MPI** é o *standard* da programação paralela em sistemas de memória distribuída. Como a memória não é partilhada os processadores têm de comunicar informação entre si. Para além destas etapas de comunicação a etapa de finalização é muito importante (e muito frequentemente esquecida na programação) já que permite desalocar as variáveis e tornar a memória disponível de novo.

Por razões de economia e engenharia estão a tornar-se comuns sistemas com múltiplos nodos (cada um com vários processadores) de memória partilhada. Estes sistemas permitem que se use simultaneamente programação com OpenMP (intra-nodo) e programação com MPI (inter-nodo).

Introdução à Programação Usando OpenMP

Esta apresentação foi dada por Kent Milfeld.

Uma das principais motivações para a paralelização é o facto de actualmente os computadores *laptops* terem vulgarmente dois *cores* e muito possivelmente no futuro poderão vir a ter quatro *cores*.

O OpenMP é o *open standard* para a programação científica paralela de sistemas SMP. Para além do uso de memória partilhada o OpenMP diferencia-se do MPI também pelo uso de directivas de compilador (activas por compilação opcional) e pelo facto de o número de *threads* (*cores*) não ter de ser à partida especificado na etapa de programação. Existem directivas próprias para Fortran e para C.

A paralelização que se consegue com este conjunto de directivas pode ser *fine-grained*, por exemplo ao nível de ciclos, ou *coarse-grained*. A paralelização de ciclos é a abordagem utilizada na grande maioria das aplicações de OpenMP.

Num sistema computacional de memória distribuída existe apenas um sistema operativo que gere a alocação de processos pelos vários processadores existentes. Caso vários processos sejam alocados ao mesmo processador dividem o tempo de uso através de *time sharing*, o qual tem grandes *overheads*.

A criação de *threads*, tarefas que correm simultaneamente resultantes de um mesmo processo (por bifurcação), permite reduzir os *overheads* do *time sharing* ao mesmo tempo que permite o acesso a memória comum, o que não é permitido em processos diferentes pelo sistema operativo. A implementação de *threads* difere de um sistema operativo para outro.

É importante notar-se que o número de threads que é possível gerar-se não está limitado pelo número de *cores* existentes na máquina. Caso sejam especificados mais que este número o sistema operativo começa a fazer *time sharing* da forma tradicional aumentando os *overheads*. Desta forma, apesar de o número de *cores* não ser limitante é importante fazer a atribuição do número de *threads* de acordo com o número de cores e os processos (externos) existentes a correr ou que se sabe terem de correr nesses *cores*. Quando diferentes *threads* tentam aceder à mesma localização de memória cria-se uma *race condition*.

As directivas OpenMP dividem-se em estruturas de controlo de paralelização (estabelecendo regiões paralelas), de divisão de trabalho pelos vários *threads*, ambiente de dados, sincronização (coordenação da execução dos *threads*), ambiente de *runtime*. Os pormenores da especificação destas directivas podem ser observados na apresentação de Kent Milfeld.

Na programação com OpenMP há que ter em conta que os *overheads* aumentam com o número de *threads*. Isto faz com que seja preferível criar uma única região paralela (em que o trabalho é

feito pelos vários *threads*) e aí serializar secções de código que por natureza devam ser processadas apenas por um *thread* do que criar diversas regiões paralelas. Desta forma, elimina-se o *overhead* da formação de equipas de *threads*, que se faz no início de cada definição de região paralela.

Na apresentação deu-se especial relevância às cláusulas referentes ao *schedule* (distribuição de processamento) dos *threads* em runtime. Salientou-se o valor da cláusula *STATIC*: com baixo *overhead* e nenhum *overhead* de sincronização, é preferível quando a carga de trabalho distribuída a cada processador é sempre igual. Caso a carga de trabalho não seja sempre igual, por exemplo dependa da iteração específica de um ciclo, uma boa opção será escolher a cláusula *DYNAMIC*.

Referiu-se também a importância da definição de variáveis privadas, através da cláusula *PRIVATE*. Isto evita que problemas de processamento quando diferentes *threads* acedem à mesma localização de memória. Por definição estas variáveis privadas têm valor indefinido à entrada e saída da região paralela, o que pode ser contornado usando as cláusulas *LASTPRIVATE* (passagem do último valor da variável privada para a variável partilhada) e *FIRSTPRIVATE* (passagem do valor da variável partilhada para as variáveis privadas à entrada da região paralela).

O OpenMP possui também algumas funções para aceder a variáveis definidas em *runtime*, como seja o número de *threads* (*omp_get_num_threads*) e o número de processadores (*omp_get_num_procs*), de modo a que o processamento possa ser optimizado para o valor desta variáveis.

Engenharia de Optimização e Desempenho para Aplicações Científicas

Esta sessão foi apresentada por Lars Koesterke.

A optimização requer um conhecimento aprofundado da arquitectura dos sistemas e do modo como os compiladores funcionam, sendo um processo iterativo. Em particular um aspecto a ter em conta é que a optimização de código serial (não paralelo) pode ser perigosa porque diminui o desempenho da paralelização.

Uma parte significativa da optimização por ser feita ao nível da compilação, sendo eventualmente possível usar opções de compilação específicas para cada subrotina/função. É importante que o código seja escrito de forma a mostrar claramente qual deve ser o funcionamento do compilador, sendo de evitar a programação orientada por objectos e apontadores, que são geralmente pontos de aumento de custos computacionais do código. É

possível especificar-se diferentes níveis de otimização a serem considerados na compilação desde a não otimização, à otimização da velocidade apenas até uma **otimização agressiva** que implica rearranjo de código como seja a transformação de ciclos.

Para otimização de código deve-se minimizar o comprimento de *stride*, isto é a distância entre elementos de uma matriz a que se acede num ciclo: um stride de 1 é o óptimo para o código ser vectorizável; um *stride* de 2 ou múltiplo de 2 é o pior caso, originando perdas de **memória cache** (memória de acesso rápido localizada no disco rígido). É importante também que os ciclos programados tenham iterações independentes sempre que possível, que possibilita ganhos com o uso de paralelização. Quando se têm ciclos múltiplos com alteração de valores de matrizes devem-se posicionar os ciclos de modo a que a primeira variável de acesso a uma matriz seja a do ciclo mais interno e a última variável seja a do ciclo mais externo

Fez-se a exposição dos vários tipos de memória existentes, dos seus tamanhos relativos e das suas diferentes larguras de banda (para acesso). Os extremos são a memória propriamente dita, que tem um tamanho da ordem dos GB mas uma pequena largura de banda, tipicamente 8 GB/s, e a memória *cache* de nível 1, com um tamanho de KB mas uma largura de banda da ordem dos 50 GB/s.

Introdução à Programação Usando MPI

Esta sessão foi apresentada por Dan Stanzione.

A paralelização por MPI é de maior complexidade de implementação que a paralelização por OpenMP. Por esta razão, só será rentável usar paralelização por MPI quando a quantidade de comunicações entre processadores é pequena relativamente ao número de linhas de código usado no processamento por cada um deles.

Enquanto que para OpenMP não existem praticamente competidores para o MPI existem algumas alternativas, como sejam as linguagens PGAS (Partitioned Global Address Space), de que são exemplos o UPC (linguagem C ao estilo MPI) e o Co-Array Fortran (linguagem Fortran ao estilo MPI).

Existem actualmente dois tipos de MPI: MPI-1, que é a implementação original, e o MPI-2, que é um acrescento ao MPI-1 (mas não uma substituição).

A programação MPI é organizada com declarações de início e finalização e declarações exercendo as comunicações entre processadores. Note-se que na programação deve-se incluir os ficheiros header, que no caso do Fortran é o mpif.f. Existem uma série de processos que comunicam entre si e que permitem organizar tarefas (*Communicators*).

A comunicação entre processadores pode ser feita de processador a processador, com o uso de comandos de envio (*send*) e recepção (*recv*), de modo colectivo, envolvendo todos os processadores na comunicação. Neste último caso incluem-se as operações de sincronização e de **broadcast** (envio de informação de um processador para todos os outros).

No caso da comunicação processador a processador se os dados a enviar/receber são de dimensão muito grande pode haver bloqueio do funcionamento do processador. É importante ter este aspecto em conta já que o mesmo código MPI pode funcionar num caso teste, de pequena dimensão, e não funcionar num caso realista, quando a escala do problema aumenta. Existem várias possibilidades de envio e recepção de dados de modo a controlar o bloqueio.

Desempenho de Escalabilidade

Esta sessão foi apresentada por Lars Koesterke.

O conceito de escalabilidade envolve a determinação da alteração do desempenho de uma solução de paralelização quando o número de processadores aumenta. Neste aspecto é importante ter em conta a Lei de Amdahl (apresentada na primeira sessão): quando a dimensão do problema aumenta tipicamente o peso da parte paralela aumenta, tornando maiores os ganhos da paralelização. Esta questão de escalabilidade é denominada de **escalabilidade forte** (*strong scaling*).

O conceito de escalabilidade pode também procurar responder à questão de qual a dimensão do problema que se poderá resolver com uma solução de paralelização com o aumento do número de processadores. Neste caso é relevante a **Lei de Gustafson**, que indica o desempenho teórico de uma aplicação com uma quantidade fixa de trabalho paralelo por processador tendo em conta a existência de um dado número de processadores. Esta questão de escalabilidade é denominada de **escalabilidade fraca** (*weak scaling*). Um exemplo de um problema de escalabilidade fraca é o caso SETI@Home, assumindo que todos os computadores pessoais envolvidos são semelhantes.

Referiu-se a importância dos **contadores de tempo** para avaliação da escalabilidade de uma solução, nomeadamente a existência de um contador de tempo para MPI.

Tanto a Lei de Amdahl como a Lei de Gustafson fornecem apenas limites teóricos para a escalabilidade. Isto faz com que seja importante fazerem-se testes práticos com dados reais. Uma boa prática é monitorizar-se o desempenho dos códigos várias vezes e considerar os **desempenhos médios**. Isto é relevante porque existem flutuações de desempenho devidas por exemplo a operações de Input/Output ou outras que usem a rede e em que o grau de utilização da largura de banda se faça sentir.

Introdução à Visualização Científica

Esta sessão foi apresentada por Paul Navrátil (pnav@tacc.utexas.edu).

A questão da visualização científica é importante porque o objectivo da visualização é aumentar o conhecimento que se tem de um dado problema. O processo de visualização é a progressão desde os dados passando por primitivas de visualização e gráficas até à visualização propriamente dita, que pode ser feita por iterações até ao objectivo da visualização ser atingido.

O tipo de visualização usado depende do objectivo da visualização. Tipicamente ela começa por se destinar ao aumento do conhecimento do próprio, depois à comunicação aos que estão próximos (ex. colegas de trabalho) ou a outras pessoas mais afastadas (ex. em comunicação em congressos e artigos).

Quando a quantidade de dados é elevada a visualização pode apenas poder ser feita usando ferramentas de visualização remotas em sistemas computacionais de grande capacidade, de que são exemplo os serviços de visualização oferecidos pelo TACC (ex. no sistema SPUR). Nestes casos de visualização remota é importante ter em conta o conceito de **latência**, isto é o tempo de propagação dos dados a visualizar através da rede. Como os dados se propagam à velocidade da luz a latência aumenta com a distância a que se está do sistema remoto.

O SPUR é também um sistema de alta performance (como o RANGER) mas para produção de imagens para visualização da informação contida em ficheiros demasiado grandes para serem copiados localmente com rapidez efectiva. Assim, o SPUR é um sistema que está na mesma rede do RANGER e que tem os programas de visualização instalados, podendo-se manipular as imagens através de ligação por VNC.