Teste de CP-PP
Uma reflexão

A correção deste 1º teste permitiu constatar 2 aspetos relevantes que contribuíram para os fracos resultados na classificação:

(i) a falta de conceitos básicos indispensáveis para a compreensão dos conteúdos da UC; e

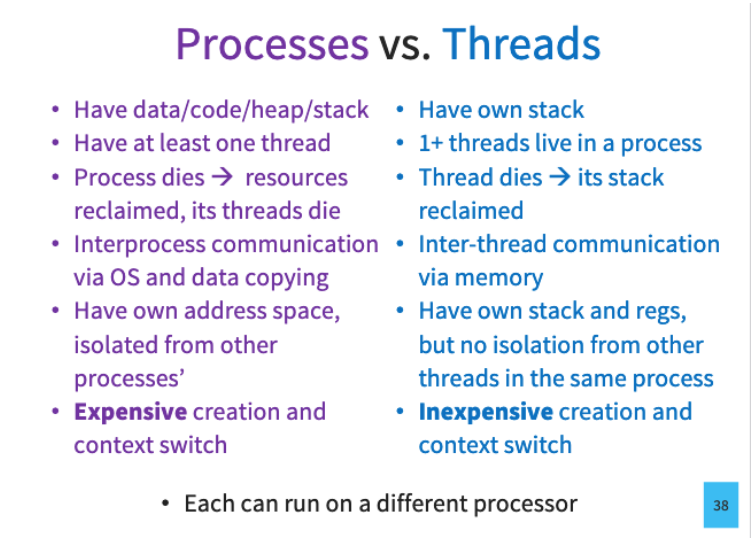
(ii) o pouco cuidado manifestado na interpretação das questões.

Antes de irmos às questões propriamente ditas do teste, uma curta reflexão sobre estes dois pontos:

(i) Alguns conceitos pertinentes

- **nó de um cluster**: é um equipamento, um computador completo, que na terminologia inglesa é referido como sendo um "*server*", para distinguir de um "*desktop*" ou de um "*laptop*"; este *server* contém normalmente um ou mais *chips multicore* (que alguns fabricantes ainda designam erroneamente por CPU), *chips* esses que estão interligados por canais de comunicação muito rápidos; como cada um desses *chips* tem o seu próprio controlador de memória com vários canais, os *chips* ficam diretamente ligados por estes canais com a RAM, construída normalmente em módulos designados por DIMM; os DIMM ligados a um dado *chip multicore* tem a memória fisicamente acessível por todos os *cores* do *chip multicore* e ainda pelos *cores* dos outros *chips multicore* que se encontram ligados a este, embora o tempo de acesso a estes circuitos de memória a partir dos *cores* dos outros *chips* seja mais longo que os tempos de acesso dos *cores* que estão no *chip* que está ligado diretamente a estes DIMMs; por isso se designa esta arquitetura de NUMA, já que o tempo de acesso de um *core* à memória não é uniforme, varia consoante a localização do circuito físico de memória a que o *core* está a aceder; esta arquitetura de um *server* tanto pode suportar computação paralela em ambiente de memória partilhada como de memória distribuída;

- **processos e threads**: se fizerem uma pesquisa no Google de "*process vs thread*" encontram muitas respostas, mas a essência é esta: "*Process means any program is in execution. Thread means segment of a process.*", ou ainda esta "*A process is a collection of code, memory, data and other resources. A thread is a sequence of code that is executed within the scope of the process. You can (usually) have multiple threads executing concurrently within the same process.*"; vejam ainda este *slide* da Cornell Univ (http://www.cs.cornell.edu/courses/cs4410/2018sp/schedule/slides/03-process-thread.pdf):



Podem ainda profundar um pouco estes dois conceitos no Wikipedia;

- "**concurrent computing**": uma pesquisa no Google de "*concurrent computing vs parallel computing*" mostra várias visões, mas a essencial é a seguinte: "*Concurrency is the task of running and managing the multiple computations at the same time. While parallelism is the task of running multiple computations simultaneously*.". Assim, quando falamos em execução concorrente de várias *threads*, isso tanto pode acontecer num *single core* (é o sistema operativo que faz a gestão de qual *thread* está a correr na UP em cada instante, criando a ilusão ao utilizador de que estão a correr ao mesmo tempo) ou as *threads* estarem distribuídas por várias *cores*, ou ainda estarem em execução em simultâneo num mesmo *core*, se o hardware deste suportar *multithreading*;

- **escalabilidade** no contexto da computação paralela: uma pesquisa no Google de "*scalability parallel performance*" mostra também várias visões, mas na essência podemos dizer que "*The scalability of a parallel algorithm on a parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors. ... For a fixed problem size, it may be used to determine the optimal number of processors to be used and the maximum possible speedup that can be obtained.*"; esta última frase está relacionado com o que se designa por "*strong scalability*" (relacionado com a lei de Amdahl), enquanto a lei de Gustafson se refere à "*weak scalability*" (vejam o *website* https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/ que compara estes dois tipos de escalabilidade); a análise da escalabilidade de algoritmos de ordenação era um dos objetivos do trabalho de grupo que foi realizado e avaliado nesta UC.

(ii) Interpretação das questões

Segue-se a análise de algumas questões do teste; de notar que não são a resolução do teste (na resolução pedia-se muito pouco texto) mas tão somente uma reflexão sobre as questões.

1. a) Nesta questão era relevante entender que a estrutura de um *server* permitia que todos os seus *cores*, quer estivessem num único *chip multicore*, quer estivessem em mais *chips* interligados, tinham acesso a uma mesma memória física e que portanto era viável implementar uma solução de um processo com múltiplos *threads*, todos eles partilhando o mesmo espaço de endereçamento que o processo-mãe; a referência a *cache* apenas é pertinente para reforçar que apenas a *cache* L3 é partilhada pelas *threads* que estão em execução em *cores* do mesmo *chip*, mas que as *caches* L1 e L2 poderão ser partilhadas pelos *threads* que estão no mesmo *core* (se o *core* suportar SMT); é a esta solução que é normalmente designada por "memória partilhada", e a utilização de OpenMP é uma das opções para implementar este modelo.

Um ambiente de trabalho com "memória distribuída" só faz sentido referir quando temos vários processos em execução paralela (simultânea) em que a comunicação entre processos é por transmissão de mensagens, já que o espaço de endereçamento de cada processo é independente dos outros; esta é a situação mais comum quando a aplicação está a correr em múltiplos *servers*, em que as memórias são independentes, embora possa haver computação em "memória distribuída" dentro de um *server* com memória física partilhada; compete ao SO garantir que cada processo não interfere com o espaço de endereçamento dos outros processos. A biblioteca do MPI e a sua lista de comandos são um instrumento precioso para auxiliar a implementação deste modelo.

1. b) Há vários slides das teóricas que referem explicitamente este pb, bem como a bibliografia recomendada para essa aula.

1. c) O enunciado indicava explicitamente que o "*código é integralmente executado no acelerador Tesla*", tendo referido antes que este era um GPU da Nvidia.

Sabendo-se que estes GPUs são pobres e extremamente limitados na execução de instruções condicionais e em operações escalares (não vetoriais), a solução lógica era colocar esta parte do código (os 20%) nos *cores* do *host* deste GPU...

2. A maioria das pessoas esqueceram-se que se os dados pretendidos não estão em *cache*, então é preciso ir buscá-los à memória externa, mas não valor a valor, tem que vir uma linha de *cache* de cada vez, linha essa de 64 *bytes*. Portanto uma vez colocada essa linha na *cache*, os 7 valores seguintes já estariam em *cache*, mas o pb repetia-se de novo em cada 8º valor que fosse preciso.

Portanto a resposta para a alínea b) deveria ser a mesma que a da alínea anterior(para cada 8º valor), já que um vetor de 8 valores ocupa precisamente uma linha de *cache*...

3. Esta questão pretendia que indicassem "*as várias alterações que tiveram que ser feitas nessa arquitetura*" e não as que deveriam ainda haver, nem ao nível do código!

Assim, o ponto de partida era "*uma arquitetura básica elementar*" (portanto sem *cache*) e respetiva lista de operações a executar, as quais incluíam ir buscar a instrução à memória (que iria levar vários ciclos de *clock* e não mostrados na figura, o que já pressupões uma execução em *pipeline* entre a busca de instruções e a sua respetiva execução), e ter de ir buscar operandos à memória (como neste caso, e sem *cache* levaria muitos ciclos de *clock* e não apenas 3 como indicado na figura).

Aqui já estão 2 melhorias que tiveram de ser implementadas.

Outras:

- em cada ciclo de *clock* é possível arrancar com a execução de mais que uma operação em simultâneo, isto é superescalaridade;

- em cada ciclo de *clock* é possível iniciar uma nova operação de `load` sem a anterior ter concluído, isto é *pipeline* na execução do `load`;

- a duplicação de unidades funcionais: há pelo menos 2 unidades para operar com inteiros, o que permite que em cada ciclo de *clock* possam ser executadas 2 operações com inteiros;

- as instruções de teste de fim do ciclo estão a ser executadas ainda antes de o `load` ter terminado, i.e., execução de instruções fora de ordem;

- uma nova iteração do ciclo é iniciada mesmo antes de se saber se o ciclo `for` já chegou ao fim ou não, isto é execução especulativa de código.

4. Com hiato crescente entre o intervalo de tempo na execução de instruções na UP e no acesso à memória externa para ir buscar instrução e dados, a solução mais óbvia para ultrapassar esta limitação na *performance* foi tirar partido da maneira com o código é executado e os dados são acedidos - a localidade espacial e temporal - criando circuitos de memória mais rápidos e colocando-os dentro dos *chips* com as UP, ou seja, construir uma hierarquia de *caches*.

Isto foi o que fizeram todos os engenheiros-arquitetos de processadores, fossem eles *chips multicore* ou GPUs.

Contudo, a aposta nos *chips multicore* foi construir *caches* cada vez maiores para evitar ao máximo acesso à memória externa, *caches* essas que vieram a ocupar uma área muito grande de silício (>50%) na construção dos *chips*.

Os arquitetos de GPUs preferiram seguir uma via diferente, não colocando *caches* demasiado grandes nos GPUs. Com a arquitetura de cada *core* de um GPU (não *CUDA-core*), que já executa operações vetoriais, com *warps* (a versão SIMT equivalente às extensões vetoriais SIMD nos processadores escalares convencionais), estes arquitetos decidiram ir bem mais longe no suporte ao *multithreading* do que aquele que já existia nos *multicores* (com apenas 2 ou 4 vias): o *hardware* de cada *core* num GPU tem registos em quantidade suficiente para em qualquer instante arrancar com um novo *warp* de entre 64 *warps* (um *multithread* de *warps*), sempre que haja necessidade de dados que não estejam em *cache*, escondendo assim as longas latências de acesso à memória.

5. Esta questão era de seleção, uma vez que se pretendia uma análise da pouca informação disponibilizada face a muita outra referida e discutida nas aulas, bem como a uma visita e estudar do site do TOP500.

Em 1º lugar convém notar que o TOP500 não reflete a posição dos mercados, como alguns afirmaram. É tão somente a criação de uma lista ordenada dos sistemas HPC mais poderosos a nível mundial, com base na execução de um determinado pacote de instruções segundo umas regras bem definidas e voluntariamente submetidos pelas entidades detentoras desses equipamentos.

Depois os gráficos apresentados referiam-se apenas às famílias de aceleradores que estavam disponíveis nos 500 sistemas submetidos para a listagem de nov-21, sendo de notar que a maioria dos sistemas não possuem aceleradores de computação (como era bem visível e foi destacado em vários outros gráficos nas aulas); aliás essa era a maior fatia nos gráficos apresentados.

Na história dos últimos anos o *manycore* da Intel já esteve na liderança dos aceleradores, mas tem vindo a perder peso nestas listagens, tendo apenas uma fatia muito pequena atualmente; o acelerador *manycore* desenvolvido pelos chineses, o Matrix 2000 (consequência da proibição dos EUA de exportação do Xeon Phi para a China), tem melhor representação em *performance* que o Xeon Phi....

O que é de facto interessante ainda é notar que 4 gerações de GPUs da Nvidia ainda têm representação significativa (Kepler, Pascal, Volta e Ampere), e que o seu impacto na *performance* global tem mais peso que o nº de sistemas que os utilizam.

O gráfico mostra ainda como os aceleradores com GPU Ampere são significativamente mais potentes que o seu antecessor, pois embora estejam instalados em menor nº de sistemas, o seu peso na *performance* global é superior.