Monografia
-
Aplicações Paralelas Híbridas (MPI, Multicore, GPU)
-
MAC 5742 2015-1

Carlos Eduardo Paladini

27 de junho de 2015

Sumário

1 Objetivo **2**

2 Introdução **2**

3 Destacando algumas tecnologias **4**

4 MPI: uma possível sinergia com OpenMP **6**

5 MPI + CUDA também é possível **11**

6 Conclusões **16**

Lista de Figuras

1 MPI em diversos hosts, com diversos cores. 6
2 MPI em diversos hosts, e OpenMP nos diversos cores de cada host. 7
3 Máquinas terra e europa rodando um processo MPI. 9
4 Máquinas terra e europa rodando um processo MPI, com 4 threads OpenMP disparadas por estes processos. 10
5 MPI em diversos hosts, com GPU's. 11

1 Objetivo

Apresentar, de forma simples e introdutória, algumas possibilidades de integração entre as diversas tecnologias de computação paralela atualmente disponíveis, buscando-se formas híbridas de paralelização, com o intuito de melhor aproveitar os recursos de hardware existentes. Serão caracterizadas e exemplificadas algumas técnicas que fazem uso de memória distribuída, memória compartilhada, e GPU.

2 Introdução

Computação paralela é uma forma de computação onde diversos cálculos (processos) são executados simultaneamente, baseando-se na idéia de dividir grandes problemas em partes menores, a serem trabalhados concorrentemente [3].

Computação paralela não é algo novo, e os esforços na busca de paralelismo, em suas diferentes formas e métodos, aparecem desde meados e final do século 20.

Inicialmente, antes do surgimento dos dispositivos VLSI(Very Large Scale Integration-dispositivos de circuito integrado combinando milhares de transistores num único chip), havia uma grande dificuldade de integrar novas funções nos circuitos integrados fabricados até então [21].

Com o advento do VLSI(décadas de 70 e 80), a possibilidade de ganho de velocidade e poder computacional através do aumento do tamanho da palavra processada, se tornou realidade. Evoluiu-se de chips de 4 bits, para 8 bits, depois 16 bits, 32 bits, e hoje temos disponível normalmente chips de 64 bits de tamanho de palavra. O tamanho de palavra determina diretamente a quantidade de informação que o processador trabalha simultaneamente [3].

Com a evolução tecnológica ocorrida desde então, e com a queda de preços nos diversos componentes de hardware, o paralelismo hoje se tornou muito popular. Atualmente, qualquer instituição, empresa, ou até mesmo pessoa física, consegue facilmente acesso a uma máquina contendo algum tipo de processamento paralelo. E além da evolução de hardware, há disponíveis muitas tecnologias de software, possibilitando em muitos aspectos o aproveitamento da capacidade de aplicações paralelas.

No entanto, colocar em prática este paralelismo, e mais ainda, desenvolver aplicações paralelas, tem se mostrado cada vez mais sofisticado e complexo. Por um lado, observa-se uma grande variedade de ferramentas disponíveis para o desenvolvedor de aplicações, o que poderia sugerir uma facilidade maior em trabalhar. No entanto, indo na direção contrária a facilitar o trabalho, essa variedade de tecnologias disponíveis aumenta consi-

deravelmente as possibilidades de explorá-las e combiná-las, intensificando-se os desafios para o desenvolvedor de aplicações.

Podemos complicar um pouco mais, quando imaginamos a possibilidade de integrar estas diferentes tecnologias. Temos por exemplo, MPI, OpenMP, GPU's, FPGA's, clusters, grids, cloud, pra citar apenas alguns nomes genéricos. Quando pensamos em combinar estas tecnologias, o cenário se mostra mais embaralhado.

Este trabalho tenta trazer alguma luz para o cenário confuso e caótico que a descrição acima provoca. Tentou-se, de forma simples e estruturada, apresentar algumas técnicas e exemplos para auxiliar o entendimento deste 'zoológico' de tecnologias, de uma maneira iniciante, na forma de um quase-tutorial "faça você mesmo".

3 Destacando algumas tecnologias

Vamos destacar neste tópico alguns cenários de hardware e software relacionados à computação paralela.

Podemos citar algumas arquiteturas de hardware atuais, onde o paralelismo pode ser evidenciado e explorado. Neste trabalho, foi dado destaque a algumas das seguintes arquiteturas:

1 - Computador multicore - onde várias unidades de execução(cores) estão embutidas num processador. Várias tarefas podem ser colocadas para execução em paralelo neste tipo de processador.

2 - Computador distribuído - os elementos de processamento(cores) estão interligados através de rede. Cada core possui sua memória local, daí vem a denominação memória distribuída para este tipo de arquitetura.

3 - Computadores em cluster - muito disseminado hoje em dia(boa parte dos computadores do TOP500 [18] são clusters). Computadores individuais são interligados por uma rede, geralmente um tipo de rede mais rápida para conexão entre os diversos componentes(nodes).

4 - Computadores em grid - leva o conceito de cluster para redes de maior abrangêngia, geralmente a internet, onde a velocidade de comunicação não é o fator mais crítico das aplicações. Um exemplo de aplicação nesta arquitetura é o projeto SETI [17], onde qualquer pessoa no planeta pode conectar seu computador pessoal para ajudar no processamento de dados de rádiotelescópios. Interligar diferentes clusters em locais geograficamente distantes também caracteriza computação em grid. Podemos citar, neste caso, o projeto GridPP [6], uma parceria de universidades britânicas e o LHC [8].

5 - Computadores multicore com dispositivos programáveis FPGA (Field Programmable Gate Array) e NoC(Network on chip)- leva o conceito de multicore a níveis maiores de flexibilidade, utilizando-se de dispositivos reconfiguráveis de acordo com a demanda. Há propostas de clusters de cores estruturados em arquiteturas de NoC programáveis [4].

6 - Computadores dotados de placas GPU, para uso de GPGPU(General-purpose computing on graphics processing units) - máquinas relativamente simples(geralmente PC's) com placas gráficas embutidas(GPU's), das quais se aproveita o grande potencial de processamento paralelo para realização de computação de alto desempenho. Os maiores fornecedores atuais destas placas são a NVIDIA [10] e a AMD [1]. Alguns dos maiores supercomputadores do TOP500 [18] são dotados de máquinas com GPU's.

Estas são algumas das possíveis arquiteturas de hardware, disponíveis para realizar computação paralela. Quando pensamos em software, há um

leque ainda maior de opções. Podemos enumerar uma lista destas tecnologias, para citar apenas algumas.

1 - OpenMP [12] - suporta programação paralela para uso de memória compartilhada entre diversos cores de processamento. É uma API disponível para linguagem C, C++ e Fortran, e é multi-plataforma, operando desde desktops até supercomputadores.

2 - MPI [9] - é uma API, que define um padrão de linguagens para realizar computação paralela através de troca de mensagens, que funciona em diferentes arquiteturas computacionais, e em diferentes linguagens, tais como C, C++, FORTRAN e Java.

3 - CUDA [2] - plataforma de computação paralela e linguagem de programação para utilização de placas gráficas(GPU) da NVIDIA [10], apoiada nas linguagens C, C++, FORTRAN.

4 - OpenCL [11]- Open Computing Language, é uma plataforma aberta, para programação paralela de processadores modernos, tais como CPU, GPU, FPGA, DSP's e outros dispositivos. Para GPU's, tem características similares ao CUDA, porém é uma proposta mais abrangente do ponto de vista de dispositivos de processamento. Permite disparar processos(kernels) para CPU, GPU, FPGA, DSP, e outros dispositivos.

5 - Intel Cilk Plus [16] - extende C e C++ para suportar paralelismo de tarefa e de dados, e aproveitar os recursos de computadores multicore e vetorial. Disponível em licença comercial [14], também possui uma extensão no frontend Clang do LLVM [15], e uma extensão no projeto GCC [13].

6 - VHDL [19] - utilizada para programação de FPGA's. Conforme citada em [20] pode ser usada como linguagem de programação paralela de propósito geral.

7 - GO - linguagem desenvolvida pelo Google, que pode ser utilizada para aproveitar as potencialidades de máquinas multicore com memória compartilhada [5].

E este arsenal de tecnologias é apenas uma parte pequena do que existe disponível em termos de tecnologias para computação paralela. Estas duas listas são bastante incompletas, servindo apenas para dar uma idéia das possibilidades a que o desenvolvedor está exposto.

Para trazer um pouco de praticidade a esta questão, este trabalho apresentou formas de conjugar algumas destas tecnologias, oferecendo um ponto de partida para aqueles que desejarem estudar mais a fundo a integração entre elas.

4 MPI: uma possível sinergia com OpenMP

Ao utilizar a tecnologia de passagem de mensagens(MPI) em máquinas com vários cores(as máquinas hoje em dia já possuem vários cores normalmente), pode-se colocar vários processos em um mesmo host, onde cada rank individualmente seria executado em um processador diferente.

Esta abordagem, inclusive, foi adotada na disciplina MAC-5742 em 2015-1o semestre do IME-USP [7], onde realizou-se experimentos com máquinas de vários cores interligadas em rede, para estudar o uso de OpenMPI.

Além da comunicação entre processos de diferentes hosts, nesta abordagem temos diversos processos se comunicando internamente no mesmo host, usando troca de mensagens do MPI. A figura a seguir esquematiza esta estratégia.

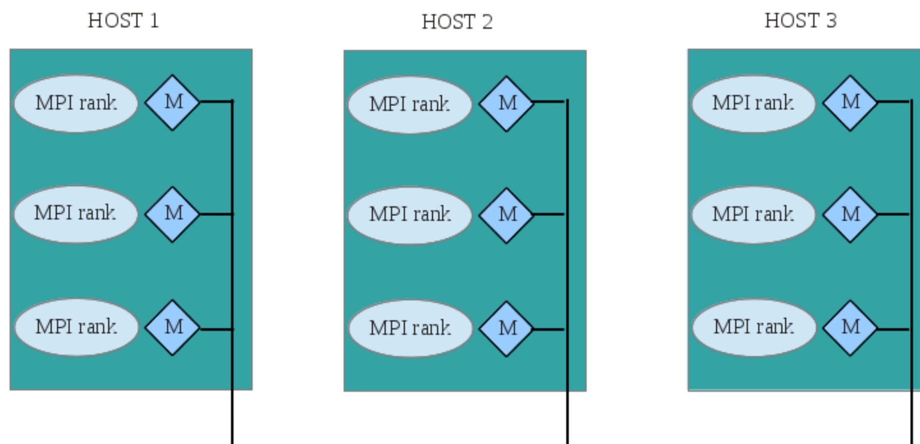

Figura 1: MPI em diversos hosts, com diversos cores.

Tal abordagem não necessariamente é a única solução, já que há processos que estão no mesmo host. Surge então a possibilidade para um novo esquema de processos.

Pode-se facilmente modificar e adaptar esta estrutura. É possível disparar um único processo MPI para cada host individualmente. E dentro de cada um destes hosts(a partir dos processos MPI individuais) poderiam ser disparadas threads em número suficiente ao número de cores presente, utilizado-se da tecnologia OpenMP. Podemos visualizar graficamente esta nova estrutura na figura a seguir.

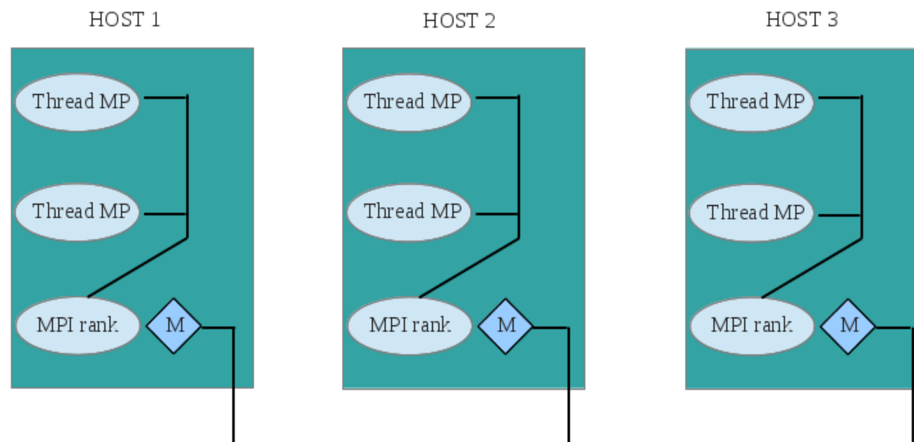

Figura 2: MPI em diversos hosts, e OpenMP nos diversos cores de cada host.

Pensando-se desta segunda forma, foi realizado um estudo prático, baseado nos códigos já desenvolvidos em outros trabalhos desta disciplina. Este estudo é apresentado a seguir.

Inicialmente, é importante destacar que este trabalho não teve a intenção de buscar ganhos de performance, nem propor uma metodologia para criar códigos híbridos. A idéia é demonstrar na prática um modesto estudo para gerar códigos híbridos.

O código utilizado para gerar a solução híbrida MPI e OpenMP foi o código para multiplicação de matrizes que utilizou inicialmente apenas MPI, no exercício da disciplina.

Podemos destacar um trecho importante deste código, onde ocorre a distribuição dos dados(matrizes) aos diversos processos. Este trecho aparece no quadro a seguir.

```
1 for (int target=1 ; target<n_procs ; target++)
2 {
3   MPI_Send(C, SIZE*SIZE, MPI_FLOAT, target, 2, MPI_COMM_WORLD);
4   MPI_Send(B, SIZE*SIZE, MPI_FLOAT, target, 0, MPI_COMM_WORLD);
5   MPI_Send(A[target*size_local],
6            SIZE*size_local, MPI_FLOAT, target, 1, MPI_COMM_WORLD);
7 }
```

Nesta parte do código fica evidente a transmissão de trechos da matriz A para cada um dos processos MPI. Esta sequência, obviamente, foi realizada apenas pelo processo com rank=0.

Por sua vez, todos os demais processos devem receber seus respectivos trechos da matriz A. Isso fica evidente no pedaço de código a seguir.

```
1   MPI_Recv( C, SIZE*SIZE, MPI_FLOAT, 0, 2, MPI_COMM_WORLD,
2                                                  &val_estado);
3   MPI_Recv( B, SIZE*SIZE, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
4                                                  &val_estado);
5   MPI_Recv( A[my_rank*size_local], SIZE*size_local,
6                       MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &val_estado);
```

De posse das matrizes necessárias para a multiplicação, todos os processos, inclusive o rank=0, passou para a etapa de multiplicação, conforme mostrado a seguir.

```
1 for (int i=my_rank*size_local; i<((my_rank+1)*size_local); i++)
2     for (int j=0 ; j<SIZE ; j++)
3         for (int k=0 ; k<SIZE ; k++)
4             C[i][j] += A[i][k]*B[k][j];
```

O leitor pode estar perguntando: como este código puro MPI, vai se transformar em um código híbrido MPI+OpenMP? A resposta para esta questão aparece a seguir, ao observar uma outra parte de código, agora sim no programa já híbrido. Veja se a mudança ficou nítida neste trecho de código.

```
1 #pragma omp parallel num_threads(4)
2 {
3   #pragma omp for collapse(3)
4   for(int i=my_rank*size_local; i<((my_rank+1)*size_local); i++)
5       for (int j=0 ; j<SIZE ; j++)
6           for (int k=0 ; k<SIZE ; k++)
7               C[i][j] += A[i][k]*B[k][j];
8
9 }
```

Realmente espantoso! A mudança foi muito simples, bastou apenas acrescentar as diretivas do OpenMP ao mesmo trecho de código de antes, do MPI puro.

Então, alguém pode perguntar, mas é só isso? Obviamente a resposta é não. No nosso exemplo, a situação é muito simples, pois o código original MPI puro já estava preparado para esta transformação, e o problema é relativamente trivial. Porém, não será sempre algo tão simples assim que o desenvolvedor irá encontrar.

Além disso, as mudanças não se resumem apenas a essa pequena alteração no código. Devemos lembrar que antes eram disparados um número de processos MPI igual ao número de cores disponíveis no total dos hosts. Devemos tomar cuidado agora, já que apenas um processo MPI pode ser chamado para cada host do nosso projeto. Estes cuidados ficam evidentes na sequência de chamadas ao nosso programa, mostrada a seguir.

```
${MPIRUN} -n 1 --machinefile hosts1.txt ep3ex3it1H >>
                              tempos_it1H_p1.h1.log
${MPIRUN} -n 2 --machinefile hosts2.txt ep3ex3it1H >>
                              tempos_it1H_p2.h2.log
```

Nas chamadas descritas acima, os arquivos hosts1.txt e hosts2.txt contém respectivamente, 1 e 2 nomes de hosts, para o nosso mpirun buscar na rede. Desta forma, cada um dos processos MPI lançados nestes 2 exemplos, serão distribuídos individualmente entre os hosts disponíveis para cada caso.

Após distribuídos os processos entre os nodes, e quando a execução dos mesmos atingir o trecho com as diretivas do OpenMP, as threads serão criadas, e serão distribuídas entre os diversos processadores dentro de cada host. Podemos observar esta situação acontecendo, através do uso do programa htop.

A seguir, são mostradas as duas situações ao rodar este programa de multiplicação. Na primeira figura, mostramos o caso simples(MPI puro) onde cada processador executa um processo MPI.

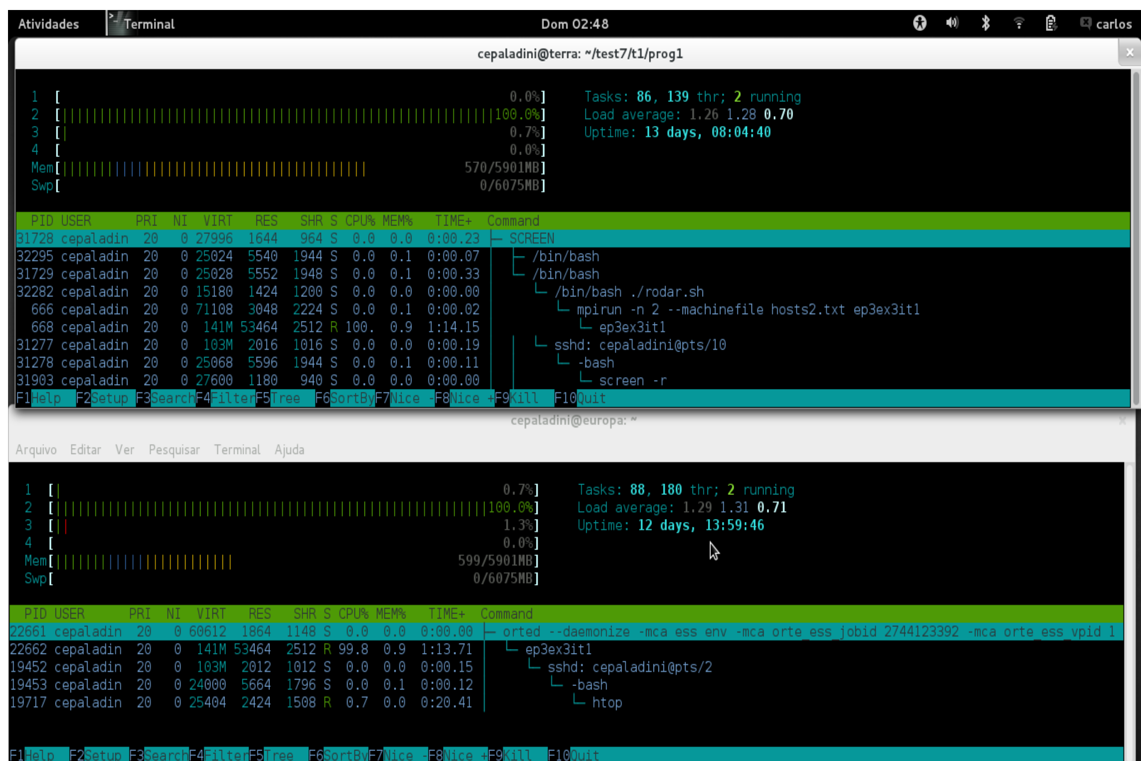

Figura 3: Máquinas terra e europa rodando um processo MPI.

Agora temos um processo MPI rodando em um host, e disparando várias threadas OpenMP para os processadores disponíveis.

Figura 4: Máquinas terra e europa rodando um processo MPI, com 4 threads OpenMP disparadas por estes processos.

Ficou assim evidenciado que é possível um processo MPI disparar várias threads OpenMP dentro de um host.

Vimos neste exemplo como fazer a alteração no código C com MPI, para obter um código C híbrido MPI + OpenMP. Vimos também como disparar este programa híbrido, tomando-se o cuidado com a forma deste disparo, para estruturar o funcionamento do programa de acordo com as máquinas disponíveis.

Um último detalhe que faltou comentar seria como trabalhar a compilação deste novo código híbrido. Lembramos que para compilar um código MPI puro utiliza-se o compilador mpicc. Para compilar um código OpenMP puro, utiliza-se o compilador gcc com uma opção de compilação em linha de comando, a opção fopenmp.

Para compilar este código OpenMP+MPI, reunimos estas duas características. Passamos a opçao fopenmp para o compilador mpicc. Tão simples assim!

5 MPI + CUDA também é possível

As mesmas máquinas que tratamos na seção anterior, utilizando MPI, podem ser dotadas de placas gráficas(GPU's). Este é o caso das máquinas utilizadas durante a disciplina, no IME-USP. Surge então uma nova possibilidade: utilizar o poder das GPU's dentro dos programas em MPI.

Podemos inicialmente lembrar da abordagem mais simples, com MPI exclusivamente, observando novamente a figura 1 . Ou seja, cada host executa diversos processos MPI.

Podemos imaginar agora alguns destes processos MPI disparando um kernel numa GPU. Esta estratégia pode ser esquematizada da seguinte forma, conforme mostra a próxima figura.

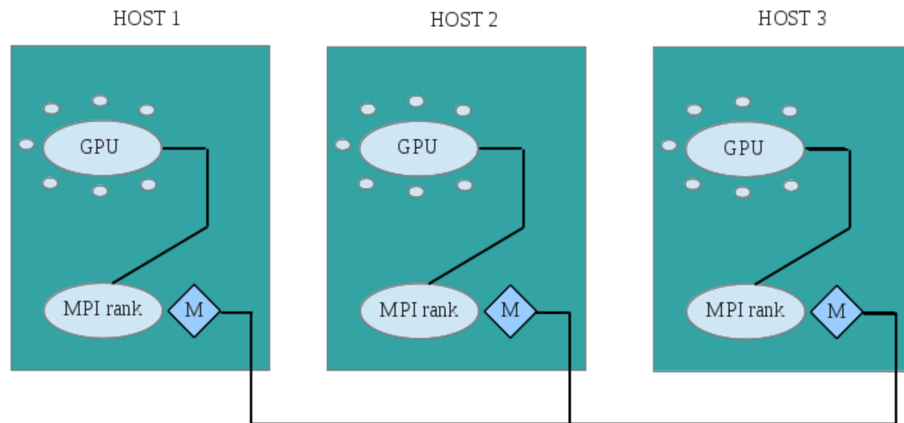

Figura 5: MPI em diversos hosts, com GPU's.

Uma primeira observação importante que podemos destacar desta figura, é que novamente devemos rodar um único processo MPI para cada host, caso todos os processos MPI tenham a intenção de disparar um kernel GPU.

Poderíamos no entanto, mudar esta estratégia, rodando diversos processos MPI num mesmo host. Porém, deveríamos tomar cuidado de deixar apenas um destes processos(em cada host) disparar um kernel na GPU por vez.

Fica muito evidente neste simples exemplo, como a escolha do método de funcionamento do meu programa no cluster vai influenciar diretamente a programação do meu aplicativo/software.

Para ilustrar esta abordagem simplificada, onde cada host executa apenas um processo MPI, que vai disparar kernels GPU, desenvolvemos um pequeno exemplo de multiplicação de matrizes(sim, de novo multiplicação)

para ilustrar alguns detalhes de programação híbrida entre MPI e CUDA.

Novamente, neste segundo programa apresentado, não tivemos a intenção de avaliar performance, nem propor uma metodologia para desenvolver código híbrido. O que pretendemos novamente, é mostrar, através de um exemplo simples, a possibilidade de computação paralela híbrida, expondo algumas dificuldades encontradas durante este simples projeto.

Podemos iniciar pela parte mais simples, destacando os hosts que foram utilizados para testar esta solução híbrida. Foram utilizadas as máquinas terra, europa, hubble e mercurio da rede eclipse.ime.usp.br, as quais são máquinas dotadas de GPU, com algumas características descritas na figura a seguir.

```
1  cepaladini@terra:~/Paca2/samples/1_Utilities/deviceQuery$
2                                                   ./deviceQuery
3  ./deviceQuery Starting...
4   CUDA Device Query (Runtime API) version (CUDART static linking)
5  Detected 1 CUDA Capable device(s)
6  Device 0: "GeForce 210"
7    CUDA Driver Version / Runtime Version          6.5 / 6.5
8    CUDA Capability Major/Minor version number:    1.2
9    Total amount of global memory:                 1023 MBytes
10                                              (1073020928 bytes)
11   ( 2) Multiprocessors, ( 8) CUDA Cores/MP:      16 CUDA Cores
12   GPU Clock rate:                                1405 MHz
13                                                 (1.40 GHz)
14   Memory Clock rate:                             550 Mhz
15   Memory Bus Width:                              64-bit
16   Maximum Texture Dimension Size (x,y,z)         1D=(8192),
17                       2D=(65536, 32768), 3D=(2048, 2048, 2048)
18   Maximum Layered 1D Texture Size, (num) layers  1D=(8192),
19                                                  512 layers
20   Maximum Layered 2D Texture Size, (num) layers  2D=(8192, 8192),
21                                                  512 layers
22   Total amount of constant memory:               65536 bytes
23   Total amount of shared memory per block:       16384 bytes
24   Total number of registers available per block: 16384
25   Warp size:                                     32
26   Maximum number of threads per multiprocessor:  1024
27   Maximum number of threads per block:           512
28   Max dimension size of a thread block (x,y,z): (512, 512, 64)
29   Max dimension size of a grid size    (x,y,z): (65535, 65535, 1)
30   Maximum memory pitch:                          2147483647 bytes
31   Texture alignment:                             256 bytes
32   Concurrent copy and kernel execution:          Yes with 1 copy
33                                                      engine(s)
34   Run time limit on kernels:                     Yes
35   Integrated GPU sharing Host Memory:            No
36   Support host page-locked memory mapping:       Yes
37   Alignment requirement for Surfaces:            Yes
38   Device has ECC support:                        Disabled
39   Device supports Unified Addressing (UVA):      No
40   Device PCI Bus ID / PCI location ID:           1 / 0
```

```
41   Compute Mode:
42      < Default (multiple host threads can use ::cudaSetDevice()
43                                        with device simultaneously) >
44 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5,
45                        CUDA Runtime Version = 6.5, NumDevs = 1,
46                                              Device0 = GeForce 210
47 Result = PASS
```

Toda a distribuição de dados(matrizes) entre os diversos processos, para o caso do programa híbrido MPI+CUDA, foi desenvolvida de forma idêntica ao programa mostrado no item anterior(MPI+OpenMP). A recuperação dos dados já processados também aconteceu de forma completamente similar ao exemplo anterior.

Porém, durante o processamento da matriz, é que surgem as diferenças do nosso exemplo com relação ao caso anteriormente estudado. Podemos observar a seguir como ficou a porção de código, onde acontece a chamada para o cálculo da matriz.

```
1     int lin_inicial = my_rank*size_local ;
2     int lin_final   = (my_rank+1)*size_local;
3     dispara_kernel_mult(A, B, C, lin_inicial, size_local, SIZE);
```

Pouco antes deste ponto do programa principal, cada host(com um único processo/rank ativo) tem sua porção de matriz para trabalhar.

Quando a execução chega neste ponto, os dados de cada host(rank) devem ser transferidos para as suas respectivas GPU's. É o que esta função 'dispara_kernel_mult()' realiza. Além dessa transferência de dados para a GPU, esta função também se encarrega de preparar e disparar os kernels para as GPU's.

Esta função 'dispara_kernel_mult()' foi colocada num programa separado do programa principal. Neste pequeno projeto, temos então dois códigos de programa:

1 - m_matrix_hib_mpi_cuda.c
2 - m_matrix_hib_cuda.cu

O primeiro programa, feito em C com biblioteca MPI, é o programa que contém todo o código de transferência de dados entre os processos MPI.

Já o segundo código implementa o kernel CUDA, bem como a função que prepara/dispara a chamada do kernel.

A seguir é demonstrada a interface destas duas funções, o kernel CUDA e a função disparadora deste kernel.

```
1  __global__ void __mult_gpu__(float *dev_A, float *dev_B,
2                               float *dev_C, int lin_inicial,
3                               int size_local, int size)
4  { ... }
5
6  extern "C" void dispara_kernel_mult(float host_A[SIZE][SIZE] ,
7              float host_B[SIZE][SIZE], float host_C[SIZE][SIZE],
8              int lin_inicial, int size_local, int size)
9  { ... }
```

Observa-se que a segunda função é extern, pois deve ser possível chamá-la a partir do programa 1.

Aqui neste ponto entramos num outro ponto importante do projeto. É aqui que acontece a fusão entre as duas tecnologias em questão, o MPI e o CUDA.

Para entender melhor, é necessário observar o nosso modesto Makefile, logo abaixo. Não foi a melhor implementação de Makefile, serviu apenas para testar rapidamente nossos códigos.

```
1  BINARIOS=multi_hib
2  OBJS=m_matrix_hib_mpi_cuda.o  m_matrix_hib_cuda.o
3  MPICC=mpicc
4  FLAGS=-std=gnu99
5  all:      ${BINARIOS}
6  multi_hib:
7          ${MPICC} ${FLAGS} -c m_matrix_hib_mpi_cuda.c
8                           -o m_matrix_hib_mpi_cuda.o
9
10         nvcc  -arch=sm_11 -c m_matrix_hib_cuda.cu
11                          -o m_matrix_hib_cuda.o
12
13         ${MPICC} m_matrix_hib_mpi_cuda.o m_matrix_hib_cuda.o
14                          -L/usr/local/cuda/lib64 -lcudart
15                          -o multi_hib
```

Conforme podemos facilmente observar neste Makefile, o nosso primeiro programa (m_matrix_hib_mpi_cuda.c), que é o responsável por toda a comunicação MPI, será compilado usando-se o mpicc.

Esse programa, conforme já foi dito anteriormente, contém uma chamada para a função 'dispara_kernel_mult()', que está no segundo programa.

Já o nosso segundo programa (m_matrix_hib_cuda.cu), que contém a preparação dos dados na GPU, bem como a definição do kernel CUDA, será compilada utilizando-se o nvcc, compilador da NVIDIA.

Por último, devemos juntar os dois arquivos objeto gerados por estes dois compiladores. Utilizamos o mpicc novamente para fazer o link dos objetos. É importante notar que ao chamar o mpicc, devemos informar o path e o nome da biblioteca do cuda, libcudart.

Com isso, temos finalmente o nosso binário, pronto para ser executado pelo mpirun. O disparo ocorre normalmente, conforme observamos no trecho de comandos a seguir.

```
1  #!/bin/bash
2  MPIRUN=mpirun
3  export LD_LIBRARY_PATH=/usr/local/cuda/lib64
4
5      sleep 13.5
6      ${MPIRUN} -n 1 --machinefile hosts1.txt  multi_hib
7
8      sleep 13.5
9      ${MPIRUN} -n 2 --machinefile hosts2.txt  multi_hib
10
11     sleep 13.5
12     ${MPIRUN} -n 4 --machinefile hosts4.txt  multi_hib
```

É evidente neste caso que foi utilizado um número de processos MPI compatível com o número de hosts descrito em cada um dos 3 casos mostrados neste disparo. Sendo assim, o arquivo hosts1.txt possui um host definido, o hosts2.txt possui 2 hosts definidos, e por último, o arquivo hosts4.txt possui 4 hosts definidos.

Com este último cuidado, o número de processos MPI fica igual ao número de hosts alocados, e em cada um deles, onde há apenas uma única GPU, não haverá competição pelo uso da mesma.

Com isso, fica demonstrada também a possibilidade de utilização das tecnologias MPI e CUDA simultaneamente.

6 Conclusões

As possibilidades de programação de sistemas paralelos híbridos são muito variadas.

Há diversas estratégias para resolver os mesmos problemas, e cada uma pode se adequar menos ou mais, de acordo com o tipo de sistema ou problema.

O estudo destas possibilidades é bastante rico, e fica a sugestão de usar as idéias deste trabalho como um ponto de partida para trabalhar este conteúdo em futuras disciplinas ligadas ao tema. Os exemplos apresentados podem servir de base para novas propostas de atividades didáticas de programação.

Nesta pesquisa ficou a impressão de que estamos caminhando para um "zoológico"de linguagens e arquiteturas, com características particulares, e que podem cooperar entre si, para aproveitar melhor os recursos de hardware disponíveis, num dado contexto, para determinada aplicação.

Também parece que estamos caminhando para uma especialização das técnicas computacionais direcionadas para cada problema. Pensando-se assim, é possível sugerir que a demanda de programadores só vai crescer, e a exigência(para o desenvolvedor) de conhecer diversas técnicas, também vai se intensificar.

O desenvolvedor, além disso, deverá cada vez mais ser um integrador de tecnologias. Esta foi a impressão causada no autor deste trabalho, após concluída a pesquisa para realização desta monografia.

Referências

[1] AMD. http://www.amd.com/.

[2] CUDA C/C++. http://www.nvidia.com.br/object/cuda_home_new_br.html.

[3] WikiPedia: Parallel Computing. https://en.wikipedia.org/wiki/Parallel_computing.

[4] Henrique Cota de Freitas. *Arquitetura de NoC programável baseada em múltiplos clusters de cores para suporte a padrões de comunicação coletiva*, 2009. http://hdl.handle.net/10183/16656?locale=pt_BR.

[5] GO-WikiPedia. https://en.wikipedia.org/wiki/Go_(programming_language).

[6] GridPP. http://www.gridpp.ac.uk/.

[7] IME-USP. http://www.ime.usp.br/.

[8] LHC. http://wlcg.web.cern.ch/.

[9] MPI-forum. http://www.mpi-forum.org/.

[10] NVIDIA. http://www.nvidia.com/.

[11] OpenCL. https://www.khronos.org/opencl/.

[12] OpenMP. http://openmp.org/wp/.

[13] GCC-Intel Cilk Plus. https://gcc.gnu.org/svn/gcc/branches/cilkplus/.

[14] Intel Cilk Plus. https://software.intel.com/en-us/intel-parallel-studio-xe.

[15] LLVM-Intel Cilk Plus. http://cilkplus.github.io/.

[16] Projeto-Intel Cilk Plus. https://www.cilkplus.org/.

[17] Projeto SETI@home. http://setiathome.ssl.berkeley.edu/.

[18] TOP500. http://www.top500.org/.

[19] VHDL. http://www.eda.org/vasg/.

[20] VHDL-WikiPedia. https://en.wikipedia.org/wiki/Parallel_programming_language.

[21] VLSI. https://en.wikipedia.org/wiki/Very-large-scale_integration.