CAP-387(2016) – Tópicos Especiais em Computação Aplicada: Construção de Aplicações Massivamente Paralelas

Aula 33: Programação Híbrida com MPI

Celso L. Mendes, Stephan Stephany

LAC / INPE

Emails: celso.mendes@inpe.br, stephan.stephany@inpe.br

Programação "Híbrida"

- Dúvida: O que é programação híbrida?
  - Múltiplas definições atualmente
- Definição de *Programação Híbrida* usada neste curso:
  - Uso de múltiplos *modelos* ou *sistemas* de programação num mesmo programa
  - Modelos de programação:
    - Abstração sobre uma forma de se programar

      Ex: troca de mensagem
  - Sistemas de programação:
    - Especificação, possivelmente acompanhada de uma implementação, de um ou mais modelo(s) de programação

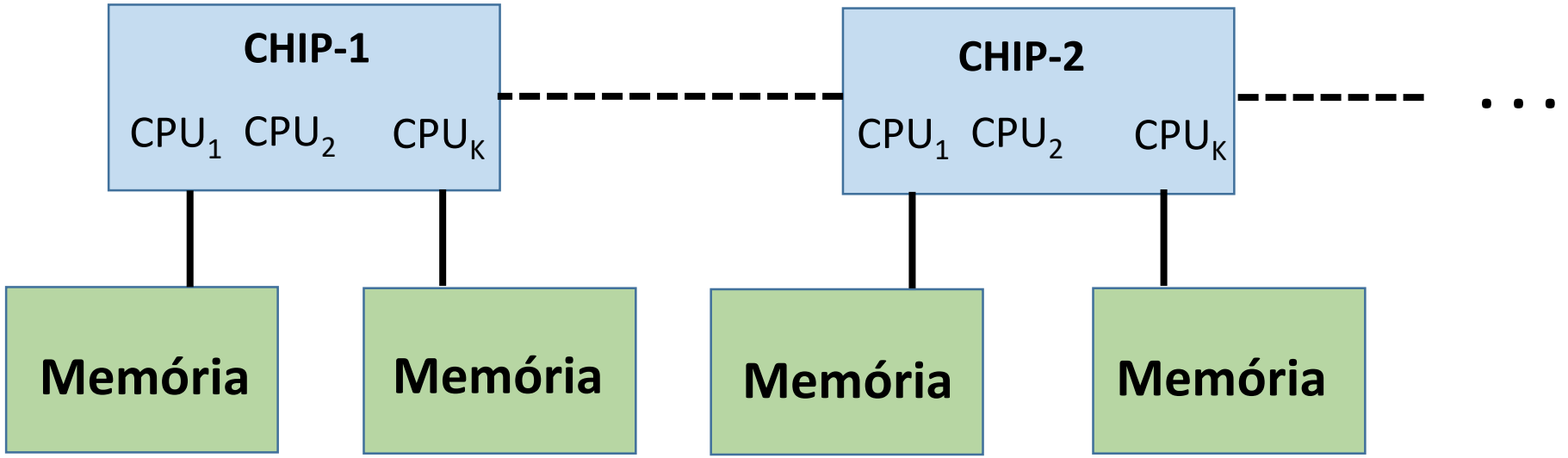
      Ex: MPI – especificação é o padrão, várias implementações atuais

Programação Híbrida - Motivação

- Tendências de sistemas atuais
  - Sistemas com múltiplos nós
  - Múltiplos núcleos por nó (em um ou mais chips)
  - Taxa de aumento do número de núcleos/nó maior que a de aumento do número de nós, atualmente
    - Sistemas massivamente paralelos: número de nós não cresce muito
    - Aumento progressivo do número de núcleos por nó
    - Tendência: uso de chips *multi-core* e *many-core*
    - Efeito: aumento no número de processadores totais deve –se principalmente ao aumento do número de núcleos/nó

Programação Híbrida - Motivação

- Nó típico num sistema moderno:
  - Um ou mais chips *multicore*
  - Um ou mais módulo(s) de memória ligado(s) a cada chip
  - Toda a memória é compartilhada por todas as CPUs no nó

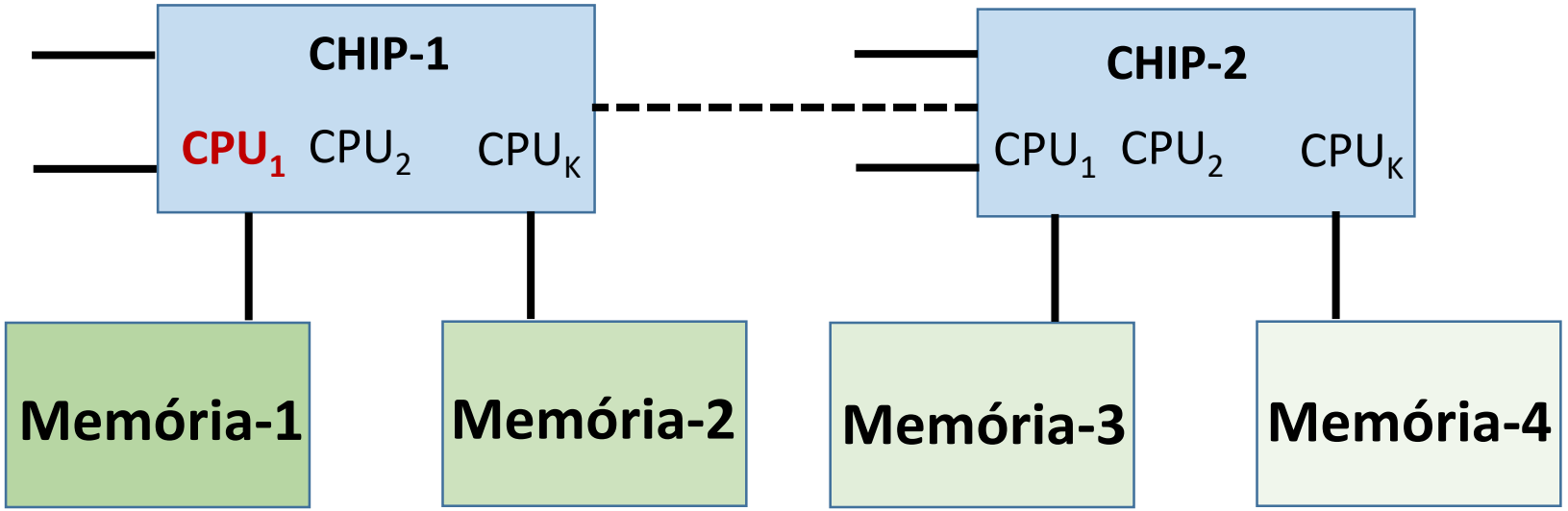


Sistema Santos Dumont

- CPU & Memória:
  - /proc/cpuinfo:
    - model name: Intel Xeon E5-2695 v2 @ 2.4 GHz (Ivy Bridge)
    - cpu cores: 12
  - /proc/meminfo:
    - MemTotal: 64 GB
  - Google (ark.intel.com): Intel Xeon E5-2695 v2
    - # of Cores: 12
    - Max # of Memory Channels: 4
    - Max Memory Bandwidth: 59.7 GB/s
    - [Package] Max CPU Configuration: 2

Programação Híbrida - Motivação

- Memória compartilhada sob a forma NUMA:
  - ➢ Mem-1 mais próxima de algumas CPUs do Chip-1 que de outras
  - Para acessos de memória da CPU_1 do Chip-1:
    - T(Mem-1) < T(Mem-2) < T(Mem-3) < T(Mem-4)



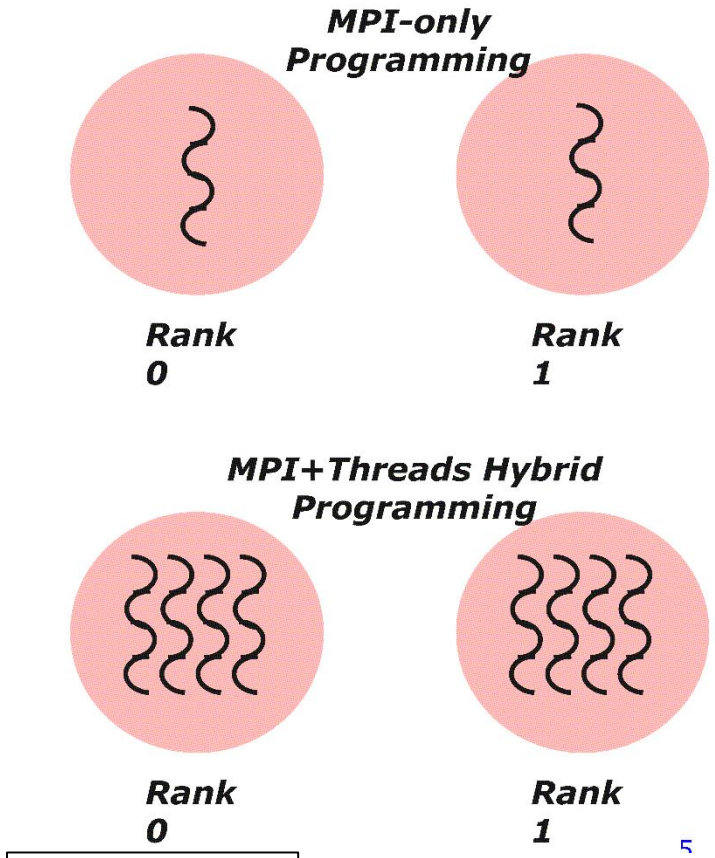
Programação Híbrida Atual

- Evolução histórica de MPI
  - Projetado, desde MPI-1, para interoperar com threads: biblioteca *thread-safe*, exceto *MPI_Probe( )* e *MPI_Iprobe( )*
  - *MPI_Init_thread* introduzido em MPI-2
  - Suporte a threads aperfeiçoado em MPI-3
- Programação Híbrida com MPI
  - Combinação de troca de mensagens com algum outro modelo
    - Ex: MPI + prog. em memória compartilhada (Pthreads, OpenMP, …)
  - Benefício: possíveis otimizações para diferentes partes da máquina
    - Otimizações intra-nó: threads, etc
    - Otimizações inter-nó: MPI, mapeamento de processos, etc

MPI com Threads

- Princípios básicos:
  - Cada processo MPI tem um *rank*
  - Cada processo pode disparar vários threads, que podem executar em CPUs ditintas, mas todos os threads compartilham o memo *rank* MPI
  - Os vários threads não são visíveis fora do processo
    - Não é possível mandar mensagens para *<rank,thread>*
    - Threads podem fazer chamadas MPI pelo seu processo
    - Cada processo tem um thrad *principal*, que cria os demais

MPI com Threads



Fonte: Bill Gropp

MPI+Threads:

- Os threads de um rank compartilham todos os objetos MPI daquele rank
- Número exato de threads por rank pode ser variável ao longo da execução
- Forma de criação dos threads não é definida pelo padrão MPI

MPI com Threads: + & -

- Vantagens:
  - + Implementação natural de comunicação sem bloqueio: enquanto um thread faz comunicação, outros ficam livres para prosseguir execução
  - + Programação com threads já é madura em sistemas de memória compartilhada, tal como num nó
  - + Com threads, há maior tolerância a operações com alta latência (se um thread bloqueia, outro pode executar)
- Desvantagens:
  - – Programação mais complexa, para garantir *thread-safety*
  - – Overhead para garantir inexistência de *race-conditions* pode levar a menor desempenho da biblioteca MPI

Padrão MPI e Threads

- Requisitos explícitos no padrão MPI
  - Chamadas à biblioteca MPI devem causar bloqueio apenas do thread que fez a chamada
  - Programas MPI corretos não devem ter múltiplos threads tentando completar (ex:com MPI_Wait) uma mesma operação MPI sem bloqueio (ex: MPI_Isend). Mas a operação pode ser iniciada/terminada por threads distintos.
  - Nível de thread exigido pela aplicação deve ser declarado em *MPI_Init* (para permitir que a biblioteca MPI "se ajuste")

Nova Versão de *MPI_Init*

- A partir de MPI-2: alternativa a *MPI_Init*:

*MPI_Init_thread(int *argc, char ***argv, int required, int *provided)*

onde *required* e *provided* são níveis de suporte a threads:

- MPI_THREAD_SINGLE: apenas um thread
- MPI_THREAD_FUNNELED: vários threads, mas apenas o thread principal pode fazer chamadas MPI
- MPI_THREAD_SERIALIZED: vários threads, mas apenas um thread pode fazer chamadas MPI de cada vez (usuário deve garantir isso)
- MPI_THREAD_MULTIPLE: vários threads, livres para fazerem chamadas MPI a qualquer momento

Nova Função *MPI_Init_Thread*

- Níveis são valores inteiros, ordenados:
  - SINGLE < FUNNELED < SERIALIZED < MULTIPLE
  - *provided* ≤ *required*:
    - Implementação válida da biblioteca MPI não é obrigada a suportar todos os níveis
- *MPI_Init* continua existindo
  - Assume MPI_THREAD_SINGLE
- Biblioteca MPI otimizada:
  - Versão da biblioteca a ser usada é escolhida em tempo real
  - Overhead proporcional ao nível de suporte a thread desejado

Modo Combinado MPI+Threads

- Forma mais natural: MPI_THREAD_FUNNELED
  - Apenas o thread principal faz chamadas MPI
  - Loops são paralelizados via threads: cada thread executa um subconjunto das iterações do loop
    - Ex: via diretivas OpenMP
    - Caso trivial: loop contém apenas computação
      - Iterações/threads podem prosseguir livremente
    - Caso não-trivial: iterações do loop chamam rotinas que invocam funções MPI
      - Problema: apenas iterações do thread principal podem fazer chamadas MPI!

Funções de Apoio a Threads

- Determinar qual o nível de suporte corrente:

*int MPI_Query_thread(int *provided)*

- Determinar se este é o thread principal:

*int MPI_Is_thread_main(int *flag)*

- Ambas podem ser usadas mesmo quando o programa utilizou *MPI_Init()*

Funções de Apoio a Threads

Exemplo de Uso:

```
int thread_level, thread_is_main;
MPI_Query_thread(&thread_level);
MPI_Is_thread_main(&thread_is_main);
If (thread_level > MPI_THREAD_FUNELLED ||
        thread_level==MPI_THREAD_FUNNELED  && thread_is_main) {
        < … chamadas MPI podem ser feitas normalmente … >
} else {
        printf("Erro: chamadas MPI proibidas neste thread\n");
        return 1;
}
```

Thread-Safety em MPI

- Problema (único) de threads em MPI-1
  - *MPI_Probe* e *MPI_Iprobe* (versões com e sem bloqueio)

*MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)*

Args. de entrada: *source, tag, comm*

Arg. de saída: *status*

→ Informa, em *status*, detalhes de uma mensagem que tenha chegado e ainda não tenha sido "recebida" (por *MPI_Recv*)

  - Potencial problema: caso onde dois ou mais threads tentam usar *status* para receber mensagens

Thread-Safety em MPI (cont.)

- Programas funcionavam bem em MPI-1 (com um único thread)

Exemplo: Alocar buffer com tamanho exatamente igual ao necessário

```
MPI_Status status;
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
MPI_Get_count(&status, MPI_INT, &msgsize);
buf = (int *) malloc(msgsize*sizeof(int));
MPI_Recv(buf, msgsize, MPI_INT, status.MPI_SOURCE, status.MPI_TAG,
comm, MPI_STATUS_IGNORE);
```

→ Se há um único thread, mensagem recebida sempre é a que havia sido detectada por *MPI_Probe*

Thread-Safety em MPI (cont.)

Possível ordem de execução do Exemplo, com mais de um thread:

Thread-1:

MPI_Probe(…,status)

buf = malloc(…)

MPI_Recv(buf,…)

. . .

Thread-2:

MPI_Probe(…,status)

buf = malloc(…)

< … atraso … >

MPI_Recv(buf,…)

. . .

- Mensagem (única) é detectada pelos dois threads – *status* é privativa
- Apenas o primeiro thread de fato recebe a mensagem que chegou

→ Segundo thread pode ficar bloqueado indefinidamente, ou receber outra msg!

Thread-Safety em MPI-3

- Antes de MPI-3:
  - Threads precisavam de alguma sincronização de modo que a sequência *MPI_Probe/MPI_Recv* fosse atômica!
- Solução adotada em MPI-3:
  - Novo objeto *MPI_Message*
  - Novas funções para detectar (*probe*) e receber mensagem, passando *MPI_Message* como argumento: *MPI_Mprobe(), MPI_Improbe(), MPI_Mrecv(), MPI_Imrecv()*
  - Apenas um thread é capaz de detectar e receber mensagem

Funções Adicionais em MPI-3

- Detectar mensagens, com bloqueio (equivalente a *MPI_Probe*)

*int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)*

- Detectar mensagens, sem bloqueio (equivalente a *MPI_Iprobe*)

*int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status*

- Receber mensagem detectada, com bloqueio

*int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)*

- Receber mensagem detectada, sem bloqueio

*int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)*

Funções Adicionais em MPI-3

Exemplo de Uso:

```
#define EXIT_TAG 65535
MPI_Message message;
MPI_Status status;
. . .
while (1) {
        MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &message, &status);
        if (status.MPI_TAG == EXIT_TAG) break;
        MPI_Get_count(&status, MPI_BYTE, &msgsize);
        buf = (char *) malloc(msgsize);
        MPI_Mrecv(buf, msgsize, MPI_BYTE, &message, &status);
        < … processar mensagem recebida … >
        free(buf);
}
```