Capítulo 3

Tuning e depuração de aplicações em Open MPI 5.0

Aleardo Manacero (UNESP)

Abstract

Open MPI is one of the most used APIs for programming parallel applications in high performance systems. The efficient application of Open MPI functionalities is rather fundamental to achieve lower costs and maximum efficiency while executing such programs. In this document we present some mechanisms offered by Open MPI 5.0 that make easier to debug and tune an application, improving the performance of our programs. The chapter starts presenting an overview of the main MPI functions to control and communicate, both collective and point-to-point, processes. We follow with the description of the Modular Component Architecture (MCA), which provides a set of options for tuning and debugging of MPI programs. In sequence several mechanisms for tuning programs, using MCA components, as well as debugging, including tools such as Valgrind, are presented. Finally, we conclude this chapter with a set of recommendations for tuning and debugging.

Resumo

Open MPI é uma das APIs mais usadas na programação de aplicações paralelas em sistemas de alto desempenho. Usar as funcionalidades do Open MPI de forma eficiente é importante para que os programas paralelos executem com o menor custo e máxima produtividade. Aqui apresentaremos mecanismos presentes no Open MPI 5.0 que facilitam os processos de depuração do código MPI e também de seu ajuste fino (tuning) de modo a aumentar a eficiência de nossos programas. O texto a seguir apresenta inicialmente um resumo das principais funções disponíveis no MPI, tanto de controle quanto de comunicação ponto-a-ponto e coletiva. Passa-se então à descrição da Modular Component Architecture (MCA), que fornece ao programador um conjunto de opções para tuning e depuração de programas MPI. Na sequência são apresentados mecanismos para tuning de programas, com o uso de componentes MCA, assim como para depuração, incluindo ferramentas como Valgrind. Por fim, fechamos o capítulo com um conjunto de recomendações finais para tuning e depuração.

MPI (Message Passing Interface) é uma API em uso para a paralelização de aplicações em uso há 30 anos, sendo ainda muito relevante para computação de alto desempenho. Seu projeto se iniciou com o objetivo de criação de um padrão para programação paralela, uma vez que naquele momento existiam várias bibliotecas trabalhando de maneiras bastante diferentes [1, 2, 3]. Como a ideia original era de se obter um padrão, o consórcio que criou o MPI permitiu que se criassem distribuições diferentes da API, como o mpich [4], LAN/MPI (descontinuado), Open MPI [5], e mesmo uma distribuição para Windows, MS-MPI [6]. Nas próximas páginas serão apresentados, em sequência, uma introdução aos comandos fundamentais do MPI, os conceitos da MCA (Modular Component Architecture) e os mecanismos presentes na distribuição openMPI para fazer o *tuning* e a depuração de aplicações MPI.

3.1. Introdução ao MPI

Um programa MPI é composto por comandos da linguagem nativa e por chamadas para funções da biblioteca do MPI. A sua execução no sistema paralelo é iniciada por um comando específico, que dispara a execução do número projetado de processos paralelos nas máquinas do sistema. Esse comando é o *mpirun* ou *mpiexec*, com a seguinte sintaxe:

```
mpirun [ options ] program [ <args> ]
```

ou ainda na seguinte forma, para programas MPMD:

```
mpirun [ global options ]
       [ local options1 ] program1 [ <args1> ]
       ...
       [ local optionsN ] programN [ <argsN> ]
```

Existem muitas opções que podem ser usadas, especificando diversas configurações para a execução do programa. Nos limitaremos nesse momento a apresentar duas delas, que indicam o número *X* de processos paralelos a serem executados e o nome de um arquivo que lista as máquinas (uma por linha) em que a execução ocorrerá, que são:

```
mpirun [ -n X ] [ --hostfile <filename> ] <program>
```

O programa em MPI deve delimitar a região de código em que comandos MPI serão utilizados. Isso implica em definir o início dessa região, com `MPI_INIT`, e seu final, com `MPI_FINALIZE`.

A chamada de `MPI_INIT` faz com que o sistema inicialize suas estruturas de controle de comunicação, definindo a variável de ambiente `MPI_COMM_WORLD`. É essa variável que armazena as informações que permitem a comunicação entre os processos paralelos.

3.1.1. Funções de controle de execução

Após a inicialização de `MPI_COMM_WORLD` é possível aos processos paralelos obter informações sobre o ambiente completo, ou mesmo modificar sua estrutura topológica. Para isso existem funções específicas de controle, para determinar por exemplo quantos processos foram disparados, ou qual o identificador daquele processo dentro da hierarquia

criada. Existem ainda comandos que permitem modificar a topologia entre os processos ou conhecer processos vizinhos nessa topologia.

A Tabela 3.1 apresenta as principais funções que permitem que os processos paralelos tenham algum controle sobre sua execução. Destacamos aqui `MPI_Comm_size`, que retorna ao processo que a chamou o número de processos paralelos que foram criados, e `MPI_Comm_rank`, que diz ao processo qual a sua identificação dentro dos comunicadores paralelos.

É importante também destacar a função `MPI_Cart_create`, que vai organizar os processos em uma topologia cartesiana. Os parâmetros para essa função determinam o número de eixos do espaço cartesiano (*dims*), sendo que cada dimensão terá até *ndims* processos. Ainda temos um parâmetro, *reorder*, usado para habilitar ou não a reorganização dos processos de forma a otimizar a topologia. Um último parâmetro, *wrap*, que indica se processos numa dimensão terão vizinhança de forma circular.

Uma última função da Tabela 3.1 que merece ser descrita é `MPI_Cart_shift`, que permite identificar os vizinhos daquele processo dentro da topologia. O parâmetro *displ* indica a distância dos vizinhos a serem identificados nos parâmetros *src* e *dst*.

3.1.2. Funções de comunicação entre processos

Com os processos criados e estruturados topologicamente, a comunicação entre eles é que viabiliza a paralelização da aplicação. Essa comunicação envolve comandos de envio/recebimento de uma mensagem, envio de mensagens de *broadcast*, etc. Apresentamos na Tabela 3.2 apenas algumas das funções existentes (as mais simples), mas deixamos claro que no MPI a comunicação pode ocorrer de várias formas, envolvendo operações síncronas ou assíncronas, com temporização, etc.

`MPI_Send` e `MPI_recv` são as funções de comunicação mais simples. É importante observar que mensagens transmitidas devem conter sempre valores de mesmo tipo (*type*), sendo que os tipos em MPI são similares aos tipos comuns em C, embora recebam o prefixo `MPI_` antes dos nomes típicos de C. O manual do Open MPI [5] apresenta com detalhes os tipos definidos.

Essa função de envio de mensagens tem como característica o fato de não retornar de sua chamada antes do conteúdo a ser transmitido ser copiado para o dispositivo de rede. Assim, `MPI_Send` é uma função bloqueante, em que o processo emissor aguarda que os dados sejam transferidos de seu *buffer* para um segundo *buffer*, que pode ser tanto um local intermediário no próprio sistema ou o de recebimento no sistema remoto. Existe, entretanto, uma função de envio não bloqueante, que é `MPI_Isend`, em que o processo é liberado antes mesmo dos dados contidos em "msg" começarem a ser transferidos para um *buffer* intermediário.

A diferença entre as duas funções está na latência e na confiabilidade da transmissão. A latência é menor quando usamos envio não bloqueante, uma vez que o processo está apto a continuar executando assim que completar a chamada de envio. Infelizmente, essa redução de latência cria a possibilidade do processo modificar os dados supostamente transmitidos antes da transferência efetivamente acontecer.

Tabela 3.1. Principais funções de controle dos processos MPI

Função	Descrição
`MPI_Comm_size(MPI_COMM_WORLD, &size)`	retorna o número de processos disparados
`MPI_Comm_rank(MPI_COMM_WORLD, &myid)`	retorna a identidade (rank) do processo
`MPI_Cart_create(MPI_COMM_WORLD,dims,dsize,wrap,reorder,cart)`	cria uma topologia cartesiana (cart) de dims dimensões, em que wrap diz se é fechada ou não e reorder diz se é possível reordenar os processos
`MPI_Cart_coords(cart, myrank, dims, coords)`	retorna as coordenadas numa topologia de dims dimensões (coords é um vetor)
`MPI_Cart_shift(cart, dim, displ, &src, &dst)`	retorna ranks dos vizinhos (a uma distância displ) na topologia
`MPI_Cart_get(cart, ndims, dims, periods, coords)`	recupera a topologia para as variáveis dims, periods e coords

Tabela 3.2. Principais funções de comunicação entre processos MPI

Função	Descrição
`MPI_Send(msg, length, type, id, tag, comm)`	envia msg para processo id
`MPI_Recv(msg, length, type, id, tag, comm, status)`	recebe msg do processo id
`MPI_Sendrecv(smsg, slength, stype, dest, stag, rmsg, rlength, rtype, source, rtag, comm, status)`	envia smsg para processo dest e recebe rmsg do processo source
`MPI_Send(msg, length, type, id, tag, comm, *request)`	envia msg para processo id sem bloquear o emissor, sinalizando a operação na variável request
`MPI_Recv(msg, length, type, id, tag, comm, status)`	recebe msg do processo id

Figura 3.1. Movimentos de dados para funções de comunicação coletiva.

	Buffer de envio				Buffer de recebimento			
	Proc1	Proc2	Proc3	Proc4	Proc1	Proc2	Proc3	Proc4
MPI_Bcast	a				a	a	a	a
MPI_Gather	a	b	c	d	a,b,c,d			
MPI_Allgather	a	b	c	d	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
MPI_Scatter	a,b,c,d				a	b	c	d
MPI_Alltoall	a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	a,e,i,m	b,f,j,n	c,g,k,o	d,h,l,p
Quando aplicável, considera-se que o processo Proc1 é o root								

MPI apresenta também funções de comunicação coletiva, apresentadas na Tabela 3.3. A mais simples delas é a de *broadcast*, `MPI_Bcast`, em que um dado processo envia um conjunto de dados para os demais processos. Nela o parâmetro `root` indica qual dos processos fará o envio aos demais. Além dela temos `MPI_Scatter` e `MPI_Gather`, que implementam funções complementares entre si. Em particular, `MPI_Scatter` faz a distribuição da mensagem a ser transmitida, de tamanho `cnt`, para os demais processos, cada um recebendo um pedaço de tamanho `rcnt`. Já `MPI_Gather` faz exatamente o contrário, isto é, junta pedaços de um *buffer* em um único *buffer* pelo processo `root`.

As duas últimas funções (Gather e Scatter) possuem as variantes `MPI_Alltoall` e `MPI_Allgather`, em que as comunicações ocorrem sem a necessidade de se definir um processo raiz. A figura 3.1, a seguir, ilustra como essas comunicações coletivas ocorrem.

Também na Tabela 3.3 temos funções que, além de disseminarem dados, fazem algum processamento sobre eles. Funções como `MPI_Reduce` e `MPI_Allreduce` executam uma operação definida sobre os dados retornados pelos processos. Essa operação pode ser de um dos tipos pré-definidos apresentados no quadro a seguir:

Operação	Significado
MPI_MAX	Máximo dos valores apresentados pelos processos
MPI_MIN	Mínimo dos valores apresentados pelos processos
MPI_SUM	Soma dos valores
MPI_PROD	Produto dos valores
MPI_LAND	E lógico
MPI_BAND	E *bitwise*
MPI_LOR	OU lógico
MPI_BOR	OU *bitwise*
MPI_LXOR	OU exclusivo lógico
MPI_BXOR	OU exclusivo *bitwise*
MPI_MAXLOC	Máximo e localização do máximo
MPI_MINLOC	Mínimo e localização do mínimo

Tabela 3.3. Principais funções de comunicação coletiva em MPI

Função	Descrição
`MPI_Bcast(msg, count, type, root, comm)`	Comunicação por broadcast, disparado pelo processo com rank root, devendo existir em todos os processos do grupo comm
`MPI_Scatter(msg, cnt, type, rmsg, rcnt, rtype, root, comm)`	envia um segmento (de tamanho cnt) de msg para cada processo em comm, que o recebe em rmsg
`MPI_Alltoall(msg, cnt, type, rmsg, rcnt, rtype, comm)`	todos processos enviam dados para todos ou outros processos
`MPI_Gather(msg, cnt, type, rmsg, rcnt, rtype, root, comm)`	recebe um segmento de tamanho cnt (msg) de cada processo e o coloca em rmsg
`MPI_Allgather(msg, cnt, type, rmsg, rcnt, rtype, comm)`	todos os processos recebem os dados de todos os outros processos, sem um processo *root*
`MPI_Reduce(operand,result,count,type,op,root,comm)`	faz como MPI_Gather, porém realizando uma operação (op) entre todos os dados enviados
`MPI_Allreduce(operand,result,count,type,op,comm)`	Comunicação por *broadcast*, disparado pelo processo com *rank root*, devendo existir em todos os processos do grupo *comm*

3.1.3. Funções miscelâneas

Além das funções mais fundamentais para comunicação e controle da execução dos processos paralelos, MPI oferece um conjunto bastante grande de funções, a maioria das quais não trataremos aqui. Isso inclui funções para sincronismo de processos, criação de grupos diferenciados de processos comunicadores, medição de tempo de execução, verificação dos canais de comunicação e até mesmo para comunicação unilateral. Algumas delas são descritas a seguir:

Barreira de sincronismo

`MPI_Barrier(group)` - em que processos pertencentes a *group* se sincronizam na barreira criada por essa função.

Verificação de canais de comunicação

`MPI_Probe(src, tag, comm. &status)` - que verifica se existe mensagem no buffer de entrada em que processos pertencentes a *group* se sincronizam na barreira criada por essa função. Observar que `MPI_Iprobe` faz a mesma operação, mas sem bloquear o processo durante a verificação.

Tratamento de tempo

`MPI_Wtime( )` - retorna o tempo atual em segundos
`MPI_Wtick( )` - retorna a precisão adotada por `MPI_Wtime`, isto é, qual o intervalo entre dois tiques do relógio

Comunicação unilateral

A comunicação unilateral permite que acessos remotos à memória, sem a necessidade de bloqueio durante a transferência de dados. Para tanto se cria uma janela de memória local que será disponibilizada para acesso remoto, usando funções específicas para transferir dados entre duas janelas definidas para acesso remoto. Apresentamos aqui apenas algumas funções, sem especificar os tipos de cada parâmetro, que podem ser facilmente encontradas nos manuais de Open MPI.

`MPI_Win_allocate(size, disp_unit, info, comm, &base, &win)` - cria uma região de memória, *win*, remotamente acessível (RMA window)

`MPI_Win_create(&base, size, disp_unit, info, comm, &win)` - define que a região *win* está exposta externamente a partir do ponteiro *base*

`MPI_Win_free(&base)` - libera a região exposta a partir do ponteiro *base*

`MPI_Put(*org_addr, org_count, org_dtype, tgt_rank, tgt_disp, tgt_count, tgt_dtype, win)` - move dados a partir de `org_addr` para a janela `tgt_rank`

`MPI_Get(*org_addr, org_count, org_dtype, tgt_rank, tgt_disp, tgt_count, tgt_dtype, win)` - move dados de `tgt_rank` para `org_addr`

3.2. Arquitetura Modular de Componentes

A Arquitetura Modular de Componentes, MCA, forma um sistema de arquivos de configuração, comandos e variáveis de ambiente, que propicia um ambiente eficiente de execução paralela com o Open MPI. A configuração de um sistema Open MPI é quase que totalmente feita pelo MCA, que forma basicamente a espinha dorsal das funcionalidades do Open MPI. O MCA é composto por projetos, *frameworks*, componentes e módulos, como descritos a seguir.

3.2.1. Projetos

Dentro da arquitetura do MCA projetos são entendidos como sendo o nível mais alto de código dentro do Open MPI, sendo definidos três projetos nessa categoria, que são:

- Open Portability Access Layer (OPAL): compreende código em baixo nível, para portabilidade de arquitetura e sistemas operacionais, sendo a camada do Open MPI mais próxima do sistema;
- Open MPI (OMPI): projeto compreendendo a API do MPI e sua infraestrutura de suporte;
- OpenSHMEM (OSHMEM): compreende a API OpenSHMEM, que suporta a manipulação de memória compartilhada para comunicação em RMA (comunicação unilateral no MPI), e sua infraestrutura de suporte.

Esses projetos formam camadas de gerenciamento do sistema. Em versões anteriores do Open MPI existia um quarto projeto, o Open MPI Runtime Environment - ORTE. O ORTE deixou de ser projeto interno para ser uma contribuição externa, que é o PMIX Runtime Reference Environment - PRRTE. As funções presentes nesse projeto cuidam do gerenciamento de processos e entrada/saída.

3.2.2. Frameworks

Os frameworks permitem gerenciar componentes específicos do MPI. Cada framework lançado em tempo de execução gerencia um tipo de componente. A lista completa de tipos de componentes manipulados pode ser obtida com a execução do comando `ompi_info`, sendo que os componentes mais usados incluem:

- Point-to-point Messaging Layer (PML), manipula componentes para implementar troca de mensagens ponto-a-ponto.
- Byte Transport Layer (BTL), usados como unidade de transporte para componentes PML.
- MPI collective algorithms (coll), para comunicação coletiva.

Figura 3.2. Saída parcial do comando ompi_info --param btl tcp --level 9

```
          MCA btl: tcp (MCA v2.1.0, API v3.1.0, Component v4.1.4)
      MCA btl tcp: ---------------------------------------------------
      MCA btl tcp: parameter "btl_tcp_links" (current value: "1", data source: default, level: 4 tuner/basic, type: unsigned_int)
      MCA btl tcp: parameter "btl_tcp_if_include" (current value: "", data source: default, level: 1 user/basic, type: string)
                   Comma-delimited list of devices and/or CIDR notation of networks to use for MPI communication (e.g.,
"eth0,192.168.0.0/16").  Mutually exclusive with btl_tcp_if_exclude.
      MCA btl tcp: parameter "btl_tcp_if_exclude" (current value: "127.0.0.1/8,sppp", data source: default, level: 1 user/basic,
type: string)
                   Comma-delimited list of devices and/or CIDR notation of networks to NOT use for MPI communication -- all devices
not matching these specifications will be used (e.g., "eth0,192.168.0.0/16").  If set to a non-default value, it is mutually
exclusive with btl_tcp_if_include.
      MCA btl tcp: parameter "btl_tcp_free_list_num" (current value: "8", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_free_list_max" (current value: "-1", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_free_list_inc" (current value: "32", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_sndbuf" (current value: "0", data source: default, level: 4 tuner/basic, type: int)
                   The size of the send buffer socket option for each connection.  Modern TCP stacks generally are smarter than a
fixed size and in some situations setting a buffer size explicitly can actually lower performance.  0 means the tcp btl will not tr
to set a send buffer size.
```

- MPI I/O (io), que trata de entrada/saída paralela de arquivos em sistemas de arquivos distribuídos.
- MPI Matching Transport Layer (MTL), usado exclusivamente como suporte para transporte de componentes PML.

O uso do comando `ompi_info` permite identificar quais parâmetros estão disponíveis no sistema e seus valores atuais. Na Figura 3.2 pode se ver parte da saída de uma execução desse comando, com dados para comunicação TCP, destacando por exemplo `btl_tcp_sndbuf` para determinar o tamanho buffer de envio mensagens, sendo que valor 0 permite ao protocolo otimizar o processo de envio.

3.2.3. Componentes

Componentes MCA formam uma coleção de códigos que implementam plugins para interfaces específicas. Esses plugins podem ser adicionados ao código a ser executado tanto em tempo de execução quanto na compilação.

3.2.4. Módulos

São instâncias de um componente, sendo que podem existir várias instâncias de um mesmo componente controladas por um único framework. Exemplos de módulos envolvem, por exemplo, módulos de TCP ponto-a-ponto.

3.2.5. Definindo parâmetros MCA

Os módulos e componentes MCA podem ter seus parâmetros definidos de várias formas. Isso permite que administradores de sistema façam ajustes finos na instalação do Open MPI para um hardware ou ambiente específico. Esses parâmetros podem ser definidos na linha de comando, variáveis de ambiente, arquivos de *tuning* de parâmetros MCA ou arquivos de configuração geral. Veremos alguns desses parâmetros nas próximas seções, mais especificamente aqueles voltados para depuração e ajuste de aplicações.

Essa flexibilidade na definição de parâmetros permite que sejam modificados em mais de um momento. Por exemplo, o administrador do sistema pode configurar parâmetros otimizados mais genéricos com o uso de variáveis de ambiente, enquanto o usuário da aplicação pode definir parâmetros adaptados à sua aplicação com arquivos de *tuning*.

Outro aspecto a ser observado é que muitos dos parâmetros MCA não são de interesse de um usuário comum. Assim, foram definidos níveis de interesse, ou uso, de parâmetros. Foram criados nove níveis, separados em três categorias com três níveis cada. Essa categorização fica evidente para o comando *ompi_info*, que apresenta informações para parâmetros que estejam em níveis no máximo igual ao valor indicado em `--level` (valor padrão é 1). As categorias e níveis são as seguintes:

1. *End user* - envolvendo parâmetros que um usuário precisa definir para sua aplicação executar corretamente. São parâmetros dos níveis 1, 2 e 3.

2. *Application tuner* - tipicamente envolve parâmetros úteis no tuning da aplicação, ou seja, no controle de recursos como tamanho de buffers.

3. *Open MPI developer* - engloba parâmetros que não se encaixam nas categorias anteriores ou que sejam especificamente projetados para depuração ou desenvolvimento do Open MPI e não de uma aplicação.

Parâmetros de linha de comando

São parâmetros definidos no comando `mpirun`. A forma geral é dada por:

```
$mpirun --mca <param_name> <value> -np 4 a.out
```

Parâmetros definidos por variáveis de ambiente

Aqui os parâmetros podem ser definidos com comandos de exportação, para variáveis do tipo `OMPI_MCA_<param_name>`. A forma geral é dada por:

```
$export OMPI_MCA_<param_name>=<value>
```

Arquivos de *tuning* de parâmetros MCA

Arquivos para definição de parâmetros podem ser criados e lidos ao disparar uma aplicação MPI. Isso implica no uso da opção `--tune` para ler o arquivo de configuração. Por exemplo, se criarmos um arquivo denominado *foo.conf*, podemos disparar a aplicação com o comando:

```
$mpirun --tune foo.conf -np 4 a.out
```

O conteúdo de um arquivo de configuração consiste de uma ou mais linhas iniciadas por $-x$ ou $-mca$, como por exemplo:

```
-x envvar1=value1 -mca param1 value1 -x envvar2
-mca param2 value2
-x envvar3
```

Arquivos de configuração geral

Arquivos de configuração geral também pode ser usados para definir parâmetros, sem a necessidade de usar tags para essa definição. Esses arquivos podem ser encontrados em dois pontos:

```
$HOME/.openmpi/mca-params.conf
$prefix/etc/openmpi-mca-params.conf
```

3.2.6. Selecionando componentes para uso em tempo de execução

Cada um dos frameworks do MCA possuem um parâmetro que permite definir quais componentes serão usados em tempo de execução. Por exemplo, o framework BTL pode ter definidos quais protocolos serão usados durante a execução da aplicação MPI. Nesse caso é possível listar quais componentes serão usados ou quais não devem ser usados. Os exemplos a seguir mostram os casos de inclusão e exclusão respectivamente:

```
$mpirun --mca btl self,sm,usnic  ...
$mpirun --mca btl ^tcp,uct  ...
```

Deve ser observado que apenas uma lista de inclusão ou de exclusão pode ser usada, pois o significado dessa seleção é de que ou se especifica quais componentes serão usados, ou se especifica quais não serão usados (usando todos os não listados).

3.2.7. Alguns parâmetros relevantes

`rmaps_default_mapping_policy`, que define como aplicações serão executadas, que pode ter valores como *nolocal* (não executa no nó local), ou *hwtcpus* (usa threads de hardware como CPUs)

`pma`, que define qual módulo de comunicação ponto-a-ponto a ser usado, entre ob1, ucx, etc.

`use-hwthread-cpus`, que permite usar *hyperthreads* como recursos de binding.

3.3. Tuning de aplicações MPI

O tuning de uma aplicação Open MPI, isto é, ajustes em como ela usará a rede, memória, comunicação coletiva, escalonamento, etc., pode ser feito com a configuração de quais componentes devem ser adicionados à aplicação. Isso significa identificar os parâmetros dentro dos frameworks do MCA que devem ser incluídos e ajustados.

3.3.1. Ajustes de transporte

Como MPI executa em uma rede, uma parte importante no processo de tuning é ajustar a forma de transmissão dos dados entre os processos comunicantes. Isso envolve os componentes BTL, PML e parâmetros de rede/transporte.

Seleção de BTLs

O Open MPI suporta diferentes modos para mover dados entre processos, como:

- **tcp**, para comunicação baseada em TCP/IP em redes Ethernet ou IP.
- **openib, ofi, ucx** (dependendo da versão) para transportes de alto desempenho (InfiniBand, RoCE, etc.).
- **sm** para comunicação entre processos no mesmo nó (memória compartilhada).
- **self** para comunicação de um processo para ele mesmo (loopback interno).

Na maior parte das vezes o Open MPI já identifica quais recursos de rede estão disponíveis e usa os mais eficientes. Por exemplo, se existir uma interface Infiniband ou RDMA, o protocolo TCP será desativado automaticamente.

Para forçar explicitamente o uso de alguma forma de transporte, como tcp por exemplo, você pode fazer:

```
$mpirun --mca btl tcp,sm,self -np 8 a.out
```

Você pode ainda configurar o limite entre mensagens curtas, transmitidas de modo otimizado sem o *handshake*, e mensagens longas acrescentando, na linha de comando anterior, o parâmetro `--mca btl_tcp_eager_limit valor`.

Já quando falamos de redes Infiniband, o Open MPI fornece um framework mais moderno do que o `openib`, que é o Unified Communication X (UCX). A vantagem do UCX é que os vários padrões para comunicação unilateral (Infiniband, RoCE, NVlink, etc) podem ser tratados de forma unificada. Deve ser observado que quando incluímos o UCX devemos, em geral, excluir explicitamente o uso do TCP, da seguinte forma:

```
$mpirun --mca pml ucx --mca btl ^tcp -np 8 a.out
```

A simples especificação de uso do UCX não implica no ajuste desejado dos vários parâmetros de comunicação. Alguns dos parâmetros que podem ser ajustados são vistos na Tabela 3.4. Destaque-se, por exemplo, `UCX_RC_TX_QUEUE_LEN`, que permite ajustar filas de acordo com a carga de comunicação. A definição desses parâmetros pode ser feita em um dos modos indicados para componentes MCA como, por exemplo, na forma de variáveis de ambiente:

```
export UCX_TLS=rc,sm,self
export UCX_NET_DEVICES=mlx5_0:1
export UCX_IB_NUM_PATHS=2
```

Observe-se que para aplicações que usam mensagens grandes, como grandes blocos de dados, recomenda-se aumentar o número de WQEs e buffers RDMA.

Observação sobre memória e RDMA: o RDMA requer que as áreas de memória de envio/recepção sejam "registradas" no adaptador InfiniBand. O Linux, por padrão, impõe limites baixos ao registro de memória, MLock, o que causa erros como:

```
warning: RLIMIT_MEMLOCK is 64 KB, cannot register memory
```

Tabela 3.4. Parâmetros ajustáveis no framework UCX.

Parâmetro	Efeito	Observação
`UCX_TLS=rc,sm,self`	Seleciona "transport layers" (rc = reliable connected, sm = shared memory).	Evita uso desnecessário de TCP.
`UCX_RC_TX_QUEUE_LEN`	Define número de Work Queue Entries (WQEs) de envio/recepção.	Aumentar para workloads intensos.
`UCX_RC_VERBS_TX_MAX_WR`	Máximo de Work Requests pendentes.	Ajustar conforme hardware.
`UCX_NET_DEVICES=mlx5_0:1`	Seleciona a HCA (Host Channel Adapter) e porta.	Útil em nós com múltiplas interfaces.
`UCX_IB_NUM_PATHS`	Número de caminhos simultâneos (multirail).	Melhora throughput em topologias com várias HCAs.

Este limite pode ser ajustado no arquivo `/etc/security/limits.conf`, ajustando-se as variáveis *hard memlock* e *soft memlock*, permitindo então a criação e uso de buffers RDMA grandes. Podemos, por exemplo, remover totalmente esses limites fazendo:

```
* hard memlock unlimited
* soft memlock unlimited
```

Tuning de operações coletivas

Comunicações coletivas como broadcast, reduce, allreduce, scatter, gather, barrier, etc., são bastante usadas em aplicações paralelas, além de terem um impacto importante no desempenho. O Open MPI oferece um framework chamado `tuned`, que permite escolhas de algoritmo de roteamento, decisões dinâmicas e tuning fino para essas funções. A escolha começa com a definição do modo de decisão, que pode ser um desses três:

1. Decisão fixa (*Fixed decision*): é o modo padrão, em que o Open MPI usa uma "árvore de decisões" interna, baseando-se no tamanho da mensagem, número de processos, entre outros, para escolher um algoritmo de comunicação coletiva ótimo.
2. Algoritmo forçado (*Forced algorithm*): em que se força o uso de um algoritmo específico para aquela operação coletiva, ignorando a árvore de decisão. Essa escolha é útil se você já conhece qual algoritmo é melhor para seu caso.
3. Decisão dinâmica (*Dynamic decision*): em que se cria um arquivo de regras mapeando o tamanho de mensagem e comunicador para o algoritmo a usar. Isso permite decisões mais calibradas sob diferentes cenários de execução.

A ativação de um desses modos de decisão é feita usando os parâmetros MCA. Por exemplo, para ativar o modo dinâmico podemos inserir os seguintes parâmetros:

```
--mca coll_tuned_use_dynamic_rules 1
--mca coll_tuned_dynamic_rules_filename /path/arq.rules
```

Sendo que em `arq.rules` você especifica entradas para cada tipo de comunicação coletiva, como "`--mca coll_tuned_alltoall_algorithm 1`", em que 1 corresponde ao algoritmo linear. Isso faz com que a cada chamada de `MPI_Alltoall` se use o algoritmo linear, sem permitir a decisão por qualquer outro algoritmo.

Quando usamos o modo forçado, as definições MCA são da forma:

```
--mca coll_tuned_allreduce_algorithm 6
--mca coll_tuned_alltoall_algorithm 3
```

Os números indicados no final de cada linha correspondem a algoritmos específicos. A lista completa pode ser obtida com a execução de "*ompi_info –param coll tuned –level 9*". Por exemplo, `MPI_Allreduce` pode ter implementação linear (1), redução + broadcast(2), *recursive doubling* (3), *ring* (4), *segmented ring* (5), Rabenseifner (6). A identificação do melhor algoritmo depende de fatores como o número de processos, a distribuição de mensagens entre processos, tamanho das mensagens, largura de banda da rede, latência, congestionamento, topologia, etc. Por isso, os modos dinâmico e forçado são mais interessantes por permitirem escolhas mais otimizadas para cada aplicação.

Como as definições aqui podem ser feitas em vários momentos, ou até mesmo não feitas, definiu-se uma ordem de prioridade para qual algoritmo será escolhido. Assim, se a escolha é feita de modo forçado, esse algoritmo é adotado mesmo que existam definições dinâmicas para aquela função de comunicação. O modo de escolha usando árvores de decisão só é usado caso não existam definições forçadas e nem definições aplicáveis no modo dinâmico.

3.3.2. Afinidade e binding de processos / threads

Além dos ajustes na forma em que os processos se comunicarão, outro aspecto importante no tuning de uma aplicação é a definição de como os processos MPI e seus threads serão atribuídos aos diferentes nós e núcleos da rede. Isso permite minimizar latências e maximizar o uso de dados em cache e também a conectividade de rede.

Isso significa estabelecer critérios para os procedimentos de mapeamento (mapping) e binding entre processos e elementos de processamento. Esses procedimentos podem ser definidos da seguinte forma:

- Mapeamento define como os processos são distribuídos entre nós e sockets, o que pode ser feito por núcleo, por nó ou ainda por socket.
- Binding (ou affinity) define a qual núcleo ou conjunto de núcleos um processo MPI ficará associado, sem permitir sua migração entre núcleos.

Vale observar que em sistemas modernos, que possuem múltiplos sockets, núcleos e hierarquia mais ampla de memória, a migração de processos entre núcleos pode degradar o desempenho por quebrar a localidade de memória. Portanto, aplicar o binding de processos acaba sendo importante.

O Open MPI oferece opções de mapping/binding diretamente no mpirun. Por exemplo, se quisermos mapeamento pelo socket e binding para os núcleos, podemos executar:

```
mpirun --map-by socket --bind-to core ...
```

Mapeamento e binding em redes Infiniband

Em redes Infiniband as operações de mapeamento e binding são ainda mais sensíveis. Nelas se deve ter informações mais detalhadas sobre a topologia real da rede, a qual pode ser obtida com o comando `lstopo --of txt`. A partir dele se pode associar manualmente processos MPI a núcleos próximos ao adaptador IB (mlx5_0, mlx5_1 etc.) para evitar tráfego pela interconexão inter-socket.

Afinidade de threads com o uso de OpenMP ou similares

Se sua aplicação combina MPI e threads, com o uso de OpenMP, por exemplo, além de fixar os processos MPI a núcleos, é importante fixar onde threads vão executar. Nesse caso podemos, por exemplo, definir as seguintes variáveis de ambiente:

```
export OMP_PROC_BIND=true
export OMP_PLACES=cores
```

Essas definições ajudam a manter threads próximas ao processo MPI em um socket físico, evitando deslocamentos através da interconexão NUMA.

3.3.3. Metodologia para benchmarking e escolha de parâmetros

As escolhas descritas nas páginas anteriores apenas fazem sentido se conduzirem, de fato, a um melhor desempenho da aplicação. Fazer esses ajustes por tentativa e erro é, obviamente, um péssimo procedimento. Além das indicações mais particulares apresentadas a seguir, recomenda-se ao leitor o estudo de técnicas para avaliação e otimização de programas, que pode ser feito de modo resumido em [7].

1. Mensure uma linha de base: execute benchmarks simples (por exemplo, ping-pong entre pares, rodadas de MPI_Bcast, MPI_Allreduce, MPI_Alltoall) com configuração padrão.

2. Profiling/tracing: colete métricas de comunicação como latência, largura de banda, tempos em coletivas, congestionamento de rede, transferência de dados, etc.

3. Varie parâmetros MCA relevantes: altere sistematicamente parâmetros como limiares eager/rendezvous, tamanhos de buffer, número de conexões, parâmetros de coletivas (force/dynamic) etc.

4. Execute benchmarks de teste de carga real (sua aplicação ou similar) para ver impacto real.

5. Use o modo dinâmico / rules file para coletivas, gerando regras baseadas nos dados medidos. O repositório ompi-collectives-tuning provê scripts para coletar dados de coletivas e gerar arquivos de tuning.

6. Itere: faça ajustes de acordo com os resultados e repita as medições, observando melhorias ou pioras de desempenho. Parâmetros que otimizam para mensagens pequenas podem degradar para mensagens grandes, e vice-versa.

7. Registre e documente as melhores configurações para cada ambiente/aplicação para uso futuro.

3.3.4. Exemplos de linha de comando com tuning

Um exemplo de comando mpirun com vários ajustes, para uma rede convencional:

```
mpirun -np 64 \
  --map-by socket --bind-to core \
  --mca pml ob1 \
  --mca btl tcp,sm,self \
  --mca btl_tcp_eager_limit 65536 \
  --mca btl_tcp_max_send_size 131072 \
  --mca coll_tuned_use_dynamic_rules 1 \
  --mca coll_tuned_dynamic_rules_filename ~/tuned.rules \
  --mca coll_tuned_allreduce_algorithm 2 \
        ./meu_programa_mpi
```

Com a execução desse comando a aplicação aplica os seguintes ajustes:

- Fixação de binding / mapping (linha 1)
- Escolha explícita de transportes (linhas 2 e 3)
- Ajustes de buffer TCP (linhas 4 e 5)
- Ativação de modo dinâmico para coletivas, com regras externas (linhas 6 e 7)
- Forçar algoritmo específico para Allreduce (linha 8)

Já as definições de variáveis de ambiente e do comando *mpirun* a seguir, fazem ajustes considerando uma rede Infiniband:

```
export UCX_TLS=rc,sm,self
export UCX_NET_DEVICES=mlx5_0:1
export UCX_RC_TX_QUEUE_LEN=512
export UCX_RC_RX_QUEUE_LEN=512
export UCX_IB_NUM_PATHS=2
```

```
mpirun -np 64 \
  --map-by socket --bind-to core \
  --mca pml ucx \
  --mca coll_tuned_use_dynamic_rules 1 \
  --mca coll_tuned_dynamic_rules_filename ~/tuned.rules \
      ./simulacao_mpi
```

Com essa configuração garantimos:

- Uso dos pacotes UCX/InfiniBand para transporte RDMA (linhas de `EXPORT`)
- Afinidade de processos a sockets (linha 1 de mpirun)
- Aplicação de regras dinâmicas para comunicação coletiva (linhas 3 e 4 de mpirun)
- Aproveita múltiplos caminhos IB (último `EXPORT`)

3.3.5. Últimas observações sobre tuning

O Open MPI oferece um bom conjunto de mecanismos para tuning de desempenho. Esses mecacnismos podem ser aplicados tanto para otimizar os processos de comunicação como também a alocação entre núcleos e nós da rede a processos e threads da aplicação (mapeamento e afinidade). O uso correto dos mecanismos de tuning permite alcançar comunicações com latência mínima e throughput máximo. A correção envolve procedimentos de medição, análise e refinamento iterativos.

3.4. Depuração de aplicações MPI

Desenvolver aplicações MPI para clusters de alto desempenho frequentemente implica lidar com erros de difícil identificação, como o uso incorreto de handles MPI, vazamentos de memória, uso incorreto de datatypes, deadlocks, abortos silenciosos, problemas de comunicação etc. O processo de depuração da aplicação é feito, no Open MPI, de modos e com métodos diversos. Esses métodos envolvem o uso de parâmetros MCA, assim como ferramentas externas para perfilamento.

A depuração de aplicações MPI não envolve, portanto, a identificação de problemas relativamente simples de execução, como ocorre com programas sequenciais. Veremos na sequência como tratar os seguintes aspectos:

- verificações em tempo de execução da API MPI (valores de argumentos, uso correto de handles, etc),
- relatórios de problemas como vazamentos de objetos MPI,
- controle sobre abortos, mensagens de erro, pilha de chamadas,
- visualização do estado interno de parâmetros de execução (MCA),
- suporte a depuradores paralelos e ferramentas externas de *profiling/tracing*

Tabela 3.5. Exemplos de parâmetros MCA para depuração

Parâmetro	Descrição / utilidade
`mpi_param_check`	Quando tem valor positivo, verifica, em tempo de execução, se os parâmetros passados para chamadas MPI são válidos. Essa verificação busca por erros como passar um NULL onde não poderia. Permite capturar erros de uso da API cedo, embora implique em perda de desempenho
`mpi_show_handle_leaks`	Quando ativado, lista handles MPI (como comunicadores, tipos de dados, requisições) que foram criados e não foram liberados após a chamada de MPI_Finalize
`mpi_no_free_handles`	Pode ser usado em conjunto com `mpi_show_handle_leaks`, permitindo identificar tentativas de uso de handles anteriormente liberados ou ainda quem fez a liberação, uma vez que apenas "marca" o handle como liberado
`mpi_show_mca_params`	Exibe (quando a inicialização do MPI ocorre) todos os parâmetros MCA e seus valores que irão afetar a execução. Isso inclui os que vieram via ambiente, linha de comando ou arquivos de configuração. Permite ter visibilidade do ambiente MPI real
`mpi_show_mca_params_file`	Se `mpi_show_mca_params` estiver definido, escreve a lista de parâmetros MCA no arquivo indicado em vez de stderr ou saída padrão
`mpi_abort_delay`	Permite a conexão de um depurador ao processo que tenha abortado, pois força que se espere o valor definido em segundos (ou até uma intervenção manual se o valor negativo) antes do processo efetivamente sair
`mpi_abort_print_stack`	Imprime um stack trace (se o sistema suportar) no momento de MPI_Abort, permitindo localizar onde ocorreu a falha, especialmente em chamadas MPI ou em código associado a MPI

3.4.1. Parâmetros MCA de depuração

É possível ativar a depuração de aplicações MPI usando parâmetros MCA, tanto por sua chamada em *mpirun*, como por variáveis de ambiente ou arquivos de configuração (como vimos na seção sobre o MCA). Na Tabela 3.5 são apresentados alguns dos parâmetros que podem ser usados para depurar códigos Open MPI.

Além desses, há variáveis para emitir mensagens de aviso quando componentes MCA não carregam, ou para exibir mais informações gerais de configuração.

3.4.2. Ferramentas de depuração externas

Além das variáveis internas de depuração, o Open MPI 5.0 suporta integração com ferramentas externas, permitindo uma inspeção mais profunda, quando for o caso. O utilitário *ompi_info*, embora não faça exatamente o trabalho de depuração, permite ver quais componentes estão instalados, que versões, caminhos e quais parâmetros MCA são su-

portados. Permite, por exemplo, listar todos os parâmetros MCA de um componente específico ao especificar um nível alto para o mesmo, como feito no comando a seguir, para parâmetros de comunicação coletiva.

```
ompi_info --param coll tuned --level 9
```

Depuradores paralelos (TotalView, DDT, outros)

Depuradores paralelos comerciais, obviamente, produzem resultados bem mais detalhados do que se pode obter com os parâmetros de depuração MCA. Uma dessas ferramentas é o TotalView [8], que é uma ferramenta de depuração largamente usada em computação de alto desempenho.

O uso de TotalView para depuração com o Open MPI demanda o uso do módulo MPIR shim para permitir que o depurador "enxergue" os processos MPI de modo consistente. TotalView pode exibir filas de mensagem, requisições pendentes do MPI, comunicadores abertos, etc, permitindo verificar o progresso das comunicações.

Outro depurador que pode ser utilizado é o DDT [9], que assim como o TotalView permite gerar informações sobre filas de requisições, filas de mensagens, deadlocks, etc.

O uso dessas ferramentas é, em geral, bastante simples. Elas possuem interfaces gráficas para que se configure a aplicação MPI. O problema com as mesmas é que, a menos de versões para experimentação, seu custo é usualmente elevado, com licenças anuais superando os 25 mil dólares.

Em uma linha semelhante, podemos usar ferramentas como HPCToolkit [10], Tau [11] ou mpiP [12], que fazem medição de desempenho (perfiladores) como auxílio na identificação de gargalos na execução. Aspectos como latência de mensagens, ocupação de banda, etc., podem ser bastante úteis no processo de depuração do código.

3.4.3. Depuração de memória em aplicações Open MPI com Memchecker e Valgrind

Em aplicações paralelas MPI, erros de memória podem ser difíceis de detectar, pois são mascarados por concorrência, podem produzir falhas não determinísticas ou ainda envolver buffers compartilhados e comunicação entre processos. Valgrind e Memchecker formam um par de ferramentos que podemos usar para depurar eventuais erros de memória.

O Valgrind é um conjunto de ferramentas de instrumentação de código que executa o binário em um ambiente simulado, verificando acessos à memória e chamadas de sistema. Já o Memchecker é uma infraestrutura interna do Open MPI que usa o Valgrind Memcheck para detectar erros de acesso à memória.

Com essas ferramentas é possível detectar vazamentos de memória, uso após liberação, validar regiões de memória em uso e verificar se buffers usados para comunicação foram devidamente inicializados.

Quando o Open MPI é compilado com suporte à depuração e valgrind, o ambiente de execução insere verificações adicionais durante chamadas MPI. Isso permite, por

Tabela 3.6. Opções úteis para aplicação do Valgrind

Opção	Descrição
`--leak-check=full`	Reporta todos os vazamentos de memória, com pilha de chamadas.
`--show-leak-kinds=all`	Mostra diferentes tipos de vazamento (definite, indirect, possible).
`--track-origins=yes`	Indica a origem de valores não inicializados.
`--gen-suppressions=all`	Gera regras automáticas de supressão para bibliotecas de sistema.
`--log-file=valgrind.%p.log`	Cria um arquivo de log separado para cada processo MPI (identificado por PID).

exemplo, validar que um buffer enviado via MPI_Send contém dados válidos, ou ainda detectar escrita fora de limites de arrays em mensagens MPI.

O suporte ao Memchecker não está habilitado por padrão. É necessário recompilar o Open MPI com as opções adequadas, como visto nos comandos a seguir, em que as flags `--enable-debug --enable-memchecker` fazem a habilitação da depuração com o uso do Memchecker.

```
$ ./configure --enable-debug --enable-memchecker
$ make -j
$ sudo make install
```

Para fazer uso dessas ferramentas é necessário instalar Valgrind previamente, com o uso de *apt*, por exemplo. Para usar Valgrind em conjunto com mpirun, basta prefixar o comando de execução:

```
$ mpirun -np 4 valgrind --leak-check=full ./meu_progr
```

Para ganhar eficiência na depuração recomenda-se algumas opções específicas para o Valgrind, vistas na Tabela 3.6. Na execução do programa compilado dessa forma, cada processo MPI_COMM_WORLD criará um arquivo valgrind denominado `<pid>.log`, com o resultado das verificações.

Uso integrado com o Memchecker

Quando o Open MPI detecta que está sendo executado sob o Valgrind, ele ativa automaticamente as verificações de memória adicionais. Essas verificações são implementadas no código-fonte da biblioteca, a partir de parâmetros MCA de comunicação coletiva e ponto-a-ponto (como ompi_coll_tuned_*, ompi_request, etc.). Assim, erros como o exemplo abaixo podem ser capturados:

```
int buf[10];
    MPI_Bcast(buf, 20, MPI_INT, 0, MPI_COMM_WORLD);
```

Sendo que Valgrind/Memchecker reportará:

```
==12345== Invalid read of size 4
==12345==    at 0x4C31C2A: memcpy (vg_replace_strmem.c:1017)
==12345==    by 0x5120B15: ompi_coll_tuned_bcast_intra_generic
==12345==  Address 0x7f... is 8 bytes after a block of size 40 alloc'd
```

Indicando que o buffer (buf) é pequeno demais para conter os 20 inteiros enviados.

Podemos ainda verificar a inicialização de buffers MPI, identificando erros como na seguinte implementação, em que se envia um vetor parcialmente inicializado. Nesse caso o Memchecker (via Valgrind) detecta valores indefinidos sendo lidos por alguma função do MPI, permitindo observar erros lógicos em inicializações parciais.

```
double A[5];
    A[0] = 1.0;  // outros elementos não inicializados
    MPI_Send(A, 5, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

Saída esperada:

```
==12450== Conditional jump or move depends on uninitialised value(s)
==12450==    at 0x52091B2: ompi_datatype_copy_content_same_ddt
```

3.4.4. Uso prático e exemplos de depuração com Open MPI 5.0

A depuração usando parâmetros MCA pode ser feita de modo bastante simples, como veremos nos exemplos a seguir:

Identificar uso incorreto de handles ou tipos de dados inválidos

Se existe a suspeita de manipulação de um comunicador após sua liberação com MPI_Comm_free, podemos executar o programa com a chamada a seguir e, poderemos constatar se existem handles não liberados no relatório gerado.

```
mpirun -np 4 --mca mpi_param_check 1 \
  --mca mpi_show_handle_leaks 1 \
  ./meu_programa
```

Entender exatamente quais parâmetros de runtime estão sendo aplicados

Podemos verificar se o ambiente de execução do MPI está como esperado executando o programa com o comando:

```
mpirun -np 4 --mca mpi_show_mca_params 1 \
  --mca mpi_show_mca_params_file /tmp/mca_params.log \
  ./meu_programa
```

Isso produzirá um log em `/tmp/mca_params.log` com todos os parâmetros MCA (de ambiente, arquivo ou linha de comando) tal como o processo de rank 0 os vê.

Diagnóstico de abortos

Se o programa aborta silenciosamente ou você quer investigar onde exatamente aborta, podemos usar:

```
mpirun -np 4 --mca mpi_abort_print_stack 1 \
  --mca mpi_abort_delay 30 \
  ./meu_programa
```

Com esse comando, quando o programa invocar MPI_Abort, ele imprimirá, se possível, o conteúdo do stack trace e esperará por 30 segundos antes de realmente terminar. Isso permite que se conecte um depurador ao processo abortado usando seu PID e hostname que aparecem na mensagem indicativa do aborto.

3.4.5. Limitações, custos e impactos no desempenho

O processo de depuração do código é, obviamente importante, mas precisa ser feito com bastante cuidado. A inserção de mecanismos de acompanhamento da execução do programa gera custos adicionais ao sistema, como:

1. **Sobrecarga de desempenho:** verificações em tempo de execução, como feito com mpi_param_check, implicam custos adicionais (checar argumentos, sobrecarga de controle), degradando o tempo de execução.
2. **Uso de memória extra:** é necessário manter estruturas extras para depuração e logging, para que se possa exibir informações sobre handles não liberados, etc.
3. **Dependência do sistema e da plataforma:** alguns mecanismos de depuração, como stack trace, dependem do sistema operacional, de suporte de símbolos de debug, se o binário foi compilado com símbolos (-g), se não foi otimizado demais.
4. **Limitação da instalação do Open MPI:** nem todos componentes, ou níveis de parâmetros, têm suporte completo à depuração, o que inviabiliza a identificação de problemas mais sutis relativos a alguns transportes de rede ou componentes MCA, que podem não entregar visibilidade de requisições pendentes, por exemplo.
5. **Interferência:** em programas MPI com threads, ou com muitos processos, ativar muita informação ou registros pode gerar congestionamento de I/O, atrasos ou até mascarar erros por mudanças no tempo de execução.

3.4.6. Últimas observações sobre depuração

O Open MPI 5.0 oferece um bom conjunto de funcionalidades para depuração. Essas funcionalidades permitem verificar o uso correto da API MPI em tempo de execução, incluindo a detecção de vazamentos de handles MPI e geração de traços para determinação de abortos. Open MPI permite ainda o uso de depuradores e perfiladores de programas paralelos, aumentando as possibilidades de visualização de problemas. O custo do processo de depuração, em termos de desempenho, é admissível em fases anteriores à fase de produção, quando se deve desativar os mecanismos de depuração.

3.5. Conclusões e recomendações finais sobre tuning e depuração

Para finalizar, apresentamos aqui uma consolidação de boas práticas e cuidados a serem tomados, tanto no processo de tuning de uma aplicação quanto de sua depuração.

3.5.1. Recomendações para tuning

Boas práticas

Quando descrevemos a metodologia para tuning apontamos alguns procedimentos importantes, que devem ser tomados como boas práticas. Além deles temos outros pontos que devem ser considerados numa lista não exaustiva, como:

1. Tenha benchmarks para diferentes classes de comunicação (pequenas, médias, grandes mensagens).
2. Evite depender exclusivamente da árvore fixa padrão para coletivas — use forced/dynamic quando pertinente.
3. Considere a topologia da rede e o agrupamento de nós no cluster.
4. Fixe processos e threads para minimizar migrações e maximizar localidade de memória.
5. Ajuste o registro de memória e configurações do sistema operacional (mlock, limites de lock de memória) para RDMA/OpenFabrics.
6. Habilitar UCX (–mca pml ucx) quando operar com Infiniband, pois é mais eficiente e suportado em clusters modernos.
7. Ainda em Infiniband use `UCX_TLS=rc,sm,self`, evitando o uso de TCP.
8. Sempre que possível teste diferentes algoritmos de comunicação coletiva, pois diferentes algoritmos são ótimos em diferentes regimes de mensagem.
9. Ajuste os tamanhos dos buffers de entrada e saída, procurando melhorar o throughput em cargas intensas.
10. Use múltiplos caminhos aproveitando HCAs múltiplas e topologias Fat-Tree.

Armadilhas

Do mesmo modo que existem boas práticas a serem seguidas, existem práticas que devem ser evitadas. Elas são:

1. Evite executar mais processos MPI do que núcleos físicos disponíveis (oversubscription). Em muitos casos, o Open MPI faz busy-waiting (“spinning”) para checar progresso de comunicação, o que pode saturar a CPU e provocar queda drástica de desempenho se houver oversubscription.

2. Não esqueça de habilitar os modos BTL *sm / self* BTLs quando customizar os modos de transporte de dados (btl).

3. Valores exagerados de buffer podem gerar uso excessivo de memória ou fragmentação.

4. Garanta que os limites de registro de memória, principalmente para RDMA, sejam adequados, pois limites baixos podem causar erros de "*memory registration*".

5. Tenha cuidado com a escolha de algoritmos, pois forçar algoritmos inadequados para um cenário específico pode piorar o desempenho.

3.5.2. Recomendações para depuração

Como já indicado, a depuração é um processo importante no desenvolvimento de aplicações MPI. Algumas recomendações finais possíveis incluem:

1. Compile com símbolos de depuração (-g) e sem remover informação de pilha, para que stack trace ou mensagens de erro sejam legíveis.

2. Testes locais pequenos primeiro, ou seja, use poucos nós/processos para reproduzir o erro, pois depurar localmente facilita a inspeção com ferramentas como gdb.

3. Habilite verificações de parâmetros MCA (`mpi_param_check`) para capturar erros de uso desde o início, nas fases iniciais de desenvolvimento.

4. Use `mpi_show_handle_leaks` para detectar vazamentos de objetos MPI — isso é importante em aplicações de longa execução ou que criam muitos comunicadores ou datatypes.

5. Documente quais parâmetros MCA foram usados — use `mpi_show_mca_params` e grave os *logs*. Isso facilita a comparação entre múltiplas execuções.

6. Combine a depuração com ferramentas externas — depuradores paralelos, tracers, profilers — para verificar também o comportamento de comunicação, não apenas erros.

7. Reduza a sobrecarga de trabalho com a aplicação em produção, desativando verificações e registros pesados quando tiver certeza que o código funciona, evitando penalidades de desempenho.

Referências

[1] FAUSEY, M. R. CPS and the Fermilab farms. In: FERMI NATIONAL ACCELERATOR LAB., BATAVIA, IL (UNITED STATES). 1992. Disponível em: <https://www.osti.gov/biblio/6946034>.

[2] CARRIERO, N.; GELERNTER, D. Linda in context. *Communications of the ACM*, ACM New York, NY, USA, v. 32, n. 4, p. 444–458, 1989.

[3] GEIST, A. et al. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. Cambridge, MA, USA: MIT Press, 1995. ISBN 0262571080.

[4] GROPP, W.; LUSK, E. Sowing mpich: a case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, v. 11, n. 2, p. 103–114, 1997.

[5] The Open MPI Community. *Open MPI v5.0.x*. 2025. `https://docs.open-mpi.org/en/v5.0.x/`. Accessado em julho de 2025.

[6] MICROSOFT. *MS - MPI v10.1.3*. 2025. Disponível em: <https://learn.microsoft.com/pt-br/message-passing-interface/microsoft-mpi>.

[7] MANACERO, A. Técnicas para análise e otimização de programas. In: *Minicursos do SSCAD 2024*. [S.l.]: SBC, 2024. cap. 6, p. 23.

[8] Perforce. *TotalView*. 2025. https://www.perforce.com/products/totalview. Acessada em outubro de 2025.

[9] Linaro Limited. *DDT - Distributed Debugging Tool*. 2025. Disponível em: <https://www.linaroforge.com/linaro-ddt>.

[10] HPCToolkit Project. *HPCToolkit*. 2025. Acessada em outubro de 2005. Disponível em: <https://hpctoolkit.org/index.html>.

[11] PERFORMANCE RESEARCH LAB. *TAU - Tuning and Analysis Utilities*. 2025. `https://www.cs.uoregon.edu/research/tau/home.php`. Acessada em outubro de 2025.

[12] LLNL. *LLNL/mpiP: A lightweight MPI profiler*. 2025. https://github.com/LLNL/mpiP. Acessada em outubro de 2025.