Capítulo 1

Introdução a Sistemas de E/S e Armazenamento Paralelo

Eduardo C. Inacio - *PPGCC/UFSC - eduardo.camilo@posgrad.ufsc.br*
Mario A. R. Dantas - *DCC/UFJF - mario.dantas@ice.ufjf.br*

Resumo

A proposta deste minicurso é oferecer uma introdução a sistemas de entrada e saída (E/S) e armazenamento paralelos voltados para ambientes de computação de alto desempenho (CAD). Por meio de uma abordagem expositiva, utilizando exemplos referentes a problemas reais, pretende-se demonstrar como o desempenho de E/S pode ser consideravelmente melhorado em aplicações distribuídas com modificações tanto na programação da aplicação quanto na configuração e utilização do sistema de armazenamento. Espera-se que, ao final deste minicurso, o participante tenha um visão geral da infraestrutura básica de armazenamento e da pilha de software de E/S encontrada nos ambientes de CAD de larga escala modernos, assim como consiga identificar e utilizar as funções básicas dos principais middlewares, bibliotecas e ferramentas para otimização de E/S.

1.1. Introdução

Tópicos relacionados a processamento continuam sendo o centro das atenções da área de computação de alto desempenho (CAD). Novas técnicas e tecnologias vêm sendo propostas nas últimas décadas visando aumentar o poder de processamento de infraestruturas computacionais de larga escala e, com isso, reduzir o tempo necessário para resolução de problemas científicos e de engenharia que dependem de sistemas de simulação e visualização. Ao mesmo tempo, a disponibilidade de infraestruturas com maior poder computacional tem incentivado a pesquisa de problemas cada vez maiores e mais complexos.

À medida que tais problemas crescem em complexidade e tamanho, têm-se observado que o volume de dados produzido e consumido também vem crescendo significativamente. Poderosos instrumentos empregados em pesquisa experimental e observacional, como aceleradores de partículas, telescópios, satélites e sequenciadores genéticos podem produzir terabytes de dados por segundo, dos quais, petabytes de dados chegam a ser armazenados anualmente [Bell et al. 2009, CERN 2016, Guzman et al. 2016]. Aplicações

de simulação computacional em áreas como cosmologia, física de altas energias, geociência e exploração e produção de petróleo podem gerar terabytes de dados de saída em uma única execução [Chen et al. 2009, Baker et al. 2014, Roten et al. 2016]. Usualmente, estes enormes conjuntos de dados são processados posteriormente por aplicações de visualização e análise para que informações relevantes para a pesquisa em questão possam ser extraídas [Mitchell et al. 2011, Nonaka et al. 2014, Dorier et al. 2016]. Vale ressaltar também a crescente sinergia entre ambientes e aplicações de CAD, Big Data e Internet das Coisas, cujos prognósticos são de demandas de dados ainda maiores, ultrapassando a escala dos yottabytes [Reed and Dongarra 2015, Radha et al. 2015, Zhao et al. 2016].

Neste contexto, o acesso, seja para leitura ou escrita, a estes grandes conjuntos de dados se apresenta como um dos principais gargalos de desempenho para estas aplicações, uma vez que a latência dos dispositivos que compõem a infraestrutura de armazenamento secundário chega a ser até seis ordens de magnitude maior do que a latência de acesso a memória principal, por exemplo [Lüttgau et al. 2018]. Consequentemente, observa-se em aplicações distribuídas que fazem uso intensivo de dados, principalmente naquelas que utilizam operações de entrada e saída (E/S) síncronas, ou seja, o processamento somente é retomado após a conclusão da operação, que o tempo utilizado para o armazenamento e recuperação de dados corresponde a uma fração considerável do tempo total de execução, podendo chegar até mesmo a superar o tempo utilizado para o processamento [Nonaka et al. 2018]. Visando atender as demandas de escalabilidade, tanto de desempenho quanto de capacidade de armazenamento, impostas por estas aplicações, infraestruturas com múltiplas camadas e sistemas de armazenamento altamente distribuídos são tradicionalmente empregados em ambientes de CAD. A Figura 1.1 apresenta um modelo de infraestrutura física de armazenamento e uma pilha de software de E/S típicos de ambientes de CAD.

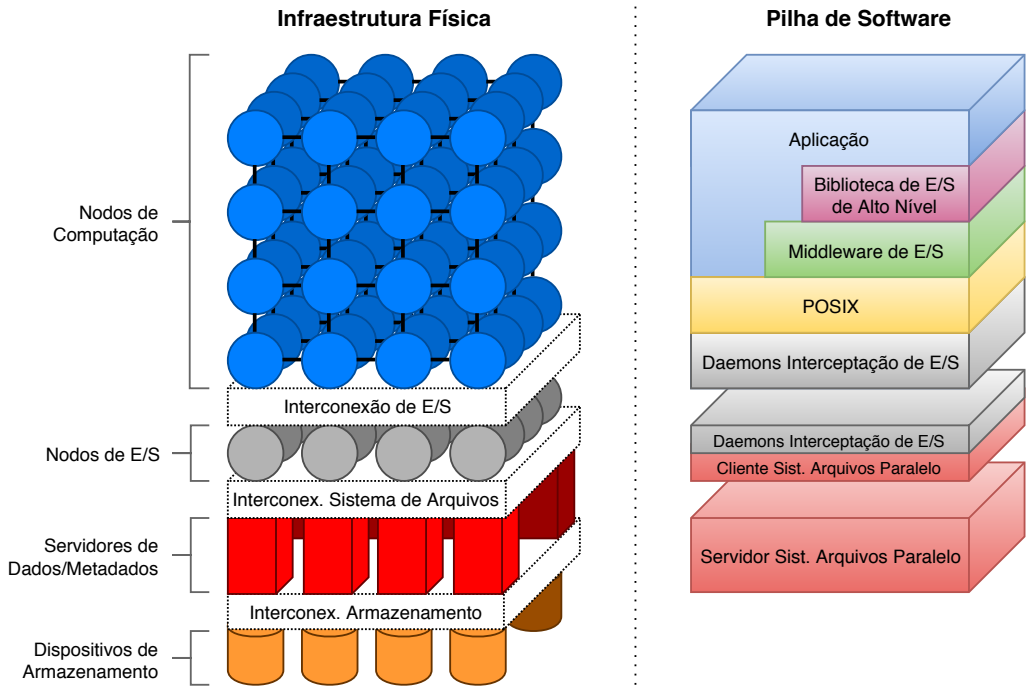


Figura 1.1. Visão geral de um modelo de infraestrutura física e uma pilha de software de E/S típicas de ambientes de CAD.

Nestes ambientes, como pode ser observado na figura, elementos de processamento e de armazenamento são bem diferenciados em termos de infraestrutura. As aplicações executam em milhares de nodos especializados, com alto poder de processamento, que se comunicam entre si utilizando redes de interconexão de alta velocidade, muitas vezes de tecnologia proprietária. Estes *nodos de computação*, em muitos casos, não possuem dispositivos de armazenamento local, tendo seu armazenamento secundário localizado inteiramente em servidores remotos. Desta forma, todos os dados necessários para a execução da aplicação, assim como toda a saída gerada por ela, precisa ser transportada entre os nodos de computação e o armazenamento secundário remoto. A aplicação pode se utilizar de diferentes bibliotecas, disponibilizadas nos nodos de computação, para realizar o acesso aos seus arquivos. As alternativas vão desde chamadas de sistema POSIX, funções coletivas de middlewares de E/S distribuídas como o MPI-IO [MPIForum 2018], até bibliotecas de alto nível como o HDF5 [The HDF Group 1997] e PNETCDF [Li et al. 2003]. Mais detalhes sobre métodos utilizados por aplicações distribuídas para armazenamento e recuperação de dados em arquivos serão apresentados na Seção 1.3.

Em grande parte dos ambientes de CAD modernos, o armazenamento secundário é formado por múltiplos discos rígidos magnéticos (hard-disk drive (HDD)), uma tecnologia madura que oferece alta durabilidade e baixo custo por byte, porém com alta latência e moderada taxa de transmissão. Novas tecnologias de armazenamento baseadas em memórias não voláteis, como solid state drive (SSD) e non-volatile random-access memory (NVRAM), vêm ganhando espaço nestes ambientes, oferecendo uma redução de até três ordens de magnitude na latência de acesso, para uma vazão até dezesseis vezes superior a dos HDDs [Lüttgau et al. 2018]. Contudo, questões de custo e características de durabilidade destes novos dispositivos ainda inviabilizam sua ampla utilização em substituição aos tradicionais HDDs no armazenamento secundário de grande porte. Uma abordagem que vem sendo utilizada em alguns ambientes e tem mostrado bons resultados consiste na utilização de SSDs e NVRAMs para a composição de uma camada intermediária entre os nodos de computação e a infraestrutura de armazenamento secundário, atuando como *nodos de E/S*. Uma das principais finalidades destes nodos é absorver as rajadas de transferência de dados dos nodos de computação, amortizando assim a latência do armazenamento secundário baseado em HDDs [Liu et al. 2012]. Assim que os dados são transferidos para os nodos de E/S, os processos da aplicação podem ser liberados para prosseguir com o processamento, enquanto os nodos de E/S transferem os dados para o armazenamento secundário propriamente dito.

Nestes ambientes de larga escala, o armazenamento secundário precisa ser altamente distribuído e escalável, de forma a suportar um grande número de requisições de E/S concorrentes, sejam elas oriundas dos processos das aplicações executando nos nodos de computação, ou dos processos coordenados pelos nodos de E/S [Prabhat and Koziol 2014]. Tradicionalmente, esse armazenamento secundário é implementado por um sistema de arquivos paralelo (SAP), como o LUSTRE [OpenSFS 2018] e o ORANGEFS [Omnibond 2018], por exemplo. Nos SAPs, como será visto com mais detalhes na Seção 1.2, os arquivos são particionados e distribuídos entre múltiplos servidores. Grandes requisições de E/S têm desempenho diferenciado nesses ambientes, uma vez que o acesso aos dados pode ser realizado de forma paralela. Adicionalmente, visando o aumento de escalabilidade, estes sistemas de arquivos, em geral, apresentam um abordagem típica de

armazenamento definido por software (software-defined storage (SDS)), separando o gerenciamento dos dados e dos metadados dos arquivos entre serviços distintos: o *servidor de dados* e o *servidor de metadados*.

Os HDDs predominam como *dispositivos de armazenamento* nos SAPs dos ambientes de CAD. Porém, diferentes formas de organização podem ser encontradas. Estes podem estar conectados diretamente aos servidores (direct-attached storage (DAS)), conectados por uma rede de interconexão de alta velocidade (storage area network (SAN)), ou na forma de um servidor de arquivos (network-attached storage (NAS)) [Troppens et al. 2009]. Tanto SAN quanto NAS permitem que vários servidores compartilhem os dispositivos de armazenamento. Uma outra forma bastante comum de organização dos dispositivos de armazenamento em ambientes de CAD é o conjunto redundante de discos independentes (redundant array of independent disks (RAID)) [Chen et al. 1994]. Essa abordagem oferece paralelismo e redundância no nível de blocos e é complementar as abordagens DAS, NAS e SAN.

Apesar da sofisticação das infraestruturas de armazenamento, tanto em termos de hardware quanto de software, encontradas nos ambientes de CAD modernos, o desempenho de E/S observado pelas aplicações pode variar consideravelmente. Esta variação se deve a um grande número de variáveis que compõem estes ambiente complexos, incluindo os padrões de acesso e a carga de trabalho da aplicação, topologia e arquitetura do sistema, configurações nas múltiplas camadas da pilha de software de E/S, propriedades dos dispositivos físicos, interferências e ruídos, entre outros [Carns et al. 2009, Lofstead et al. 2010, Inacio et al. 2015a, Inacio et al. 2015b, Herbein et al. 2016, Inacio et al. 2017a]. Uma abordagem muito empregada em pesquisas focadas no desempenho de sistemas de E/S e armazenamento paralelos é a caracterização do impacto de diferentes variáveis no desempenho de E/S observado pelas aplicações [Inacio and Dantas 2014, Boito et al. 2018]. Tais trabalhos de pesquisa se utilizam de ferramentas especializadas para a avaliação de desempenho de sistemas de E/S e armazenamento paralelos, como MPI-TILE-IO [ANL 2002], INTERLEAVED-OR-RANDOM (IOR) [Shan et al. 2008] e IOR-EXTENDED (IORE) [Inacio and Dantas 2018b]. Mais detalhes sobre essas ferramentas serão apresentados na Seção 1.4.

1.2. Sistemas de Arquivos Paralelos

Sistemas de arquivos paralelos (SAPs) como o LUSTRE [Braam and Schwan 2002, OpenSFS 2018], ORANGEFS [Moore et al. 2011, Omnibond 2018], SPECTRUM SCALE [IBM 2018], CEPH [Weil et al. 2006], e PANASAS FILE SYSTEM (PANFS) [Nagle et al. 2004], são sistemas de arquivos distribuídos especializados, focados em alto desempenho de E/S e escalabilidade horizontal. Embora variações de projeto e implementação possam ser observadas entre os diferentes sistemas, em geral, os SAPs são compostos por três componentes principais: módulo cliente, servidor de dados e servidor de metadados. A Figura 1.2 ilustra a organização tradicional destes componentes.

O *módulo cliente* consiste basicamente de uma biblioteca ou um módulo de software executando nos nodos de computação ou de E/S, que implementa o protocolo de comunicação com o SAP. Em SAPs compatíveis com POSIX, por exemplo, o módulo cliente intercepta as requisições de E/S das aplicações e realiza as operações necessárias,

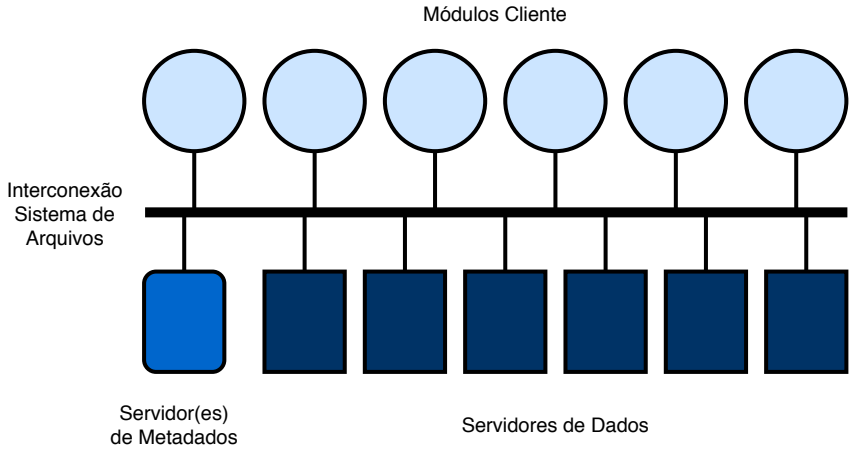


Figura 1.2. Principais componentes de um SAP genérico.

utilizando funções específicas do SAP, para o atendimento das requisições. O *servidor de metadados*, como o nome sugere, gerencia a estrutura de diretórios e mantém atualizadas as informações sobre os arquivos armazenados no sistema, como permissões, proprietário, datas de criação e atualização, localização, entre outras. Em algumas implementações de SAPs, o gerenciamento dos metadados é centralizado, com um único servidor ativo por sistema de arquivos, enquanto em outros, o gerenciamento dos metadados também é distribuído. Por fim, o conteúdo propriamente dito dos arquivos é mantido pelos *servidores de dados*. Cada servidor de dados, em geral, recebe apenas alguns fragmentos de cada arquivo, de forma que, para acessar todo o conteúdo de um arquivo, vários servidores de dados são acessados.

O processo de *particionamento e distribuição de arquivos* é uma das principais características de um SAP, que o diferencia de outros sistemas de arquivos distribuídos. A Figura 1.3 apresenta um exemplo do processo de particionamento e distribuição de um arquivo de 400 KiB em um SAP com cinco servidores de dados. É possível observar na figura que o arquivo estende-se ao longo de uma *faixa* de servidores de dados. Basicamente, dois parâmetros controlam o processo de particionamento e distribuição de arquivos em um SAP: a largura da faixa e o tamanho dos fragmentos da faixa. A *largura da faixa* (*stripe width*) define quantos servidores de dados serão utilizados para se armazenar o conteúdo de um arquivo. O *tamanho dos fragmentos da faixa* (*stripe size*) estabelece o tamanho do bloco contínuo de dados do arquivo que será armazenado em cada servidor.

No exemplo da Figura 1.3, uma largura de faixa igual a quatro e um tamanho de fragmento igual a 64 KiB foram utilizados. Primeiramente, observa-se que o arquivo é fragmentado em partes de tamanhos iguais, conforme o tamanho de fragmento estabelecido, com exceção do fragmento final, que apresenta os últimos 16 KiB do arquivo. O primeiro fragmento é armazenado no primeiro servidor, o segundo fragmento no segundo servidor, e assim por diante até que o quarto fragmento é armazenado no quarto servidor, atingindo o limite da largura de faixa. O quinto fragmento, então, é armazenado no primeiro servidor novamente, e o processo continua até que todos os fragmentos do arquivo

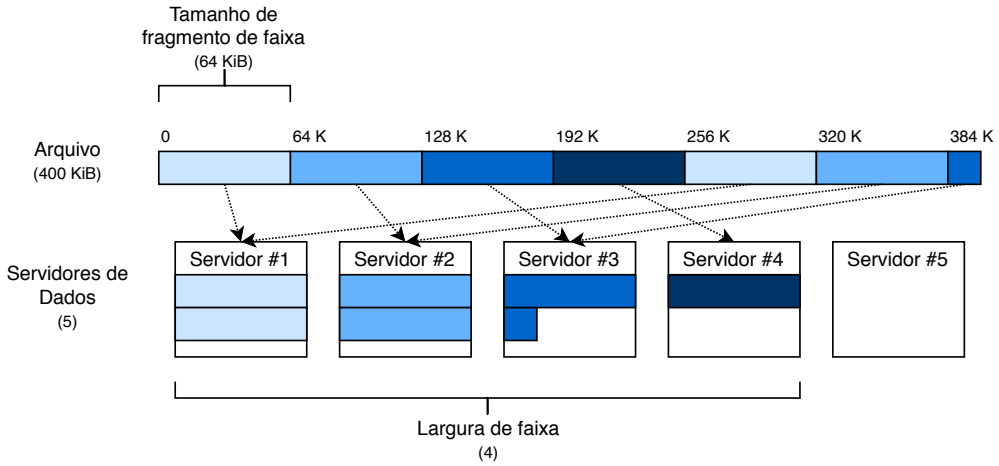


Figura 1.3. Exemplo de particionamento e distribuição de um arquivo de 400 KiB em um SAP com 5 servidores de dados, tamanho de fragmento de faixa de 64 KiB e largura de faixa igual a 4.

sejam armazenados. No caso deste arquivo ser estendido posteriormente para além dos 400 KiB, os próximos bytes seriam armazenados no *Servidor #3*, neste exemplo, até que os 48 KiB restantes do último fragmento fossem completados.

Apesar de a descrição sugerir uma sequencialidade de eventos, este mecanismo empregado pelos SAPs permite que operações de leitura e escrita possam ser realizadas de forma paralela, aumentando a vazão do sistema. Ainda sobre o exemplo da Figura 1.3, supondo uma requisição de leitura para 400 KiB do arquivo, cada servidor de dados enviaria, em paralelo, todos os fragmentos contidos nele. Em teoria, o tempo de resposta para tal requisição de leitura seria quatro vezes inferior ao tempo de resposta obtido com o arquivo armazenado em um único servidor de dados. Tal observação leva naturalmente à conclusão de que quanto maior a largura da faixa (mais servidores de dados), melhor o desempenho das requisições de E/S. Contudo, na prática, custos de comunicação com múltiplos servidores combinados ao overhead do processo de particionamento e distribuição fazem com que o desempenho estabilize a partir de uma determinada largura de faixa, fazendo com que a utilização de mais servidores de dados não resulte em incrementos de desempenho [Inacio et al. 2015a].

Dada esta característica prática relacionando a distribuição do arquivo com o desempenho das operações de acesso aos dados, layouts de distribuição costumam ser explorados em diferentes situações. Estes layouts de distribuição podem ser classificados basicamente em três grupos: horizontal, vertical e bidirecional [Song et al. 2012]. A Figura 1.4 ilustra a distribuição de quatro arquivos, com quatro fragmentos cada, segundo cada um destes três layouts de distribuição, em um SAP com quatro servidores de dados.

No *layout horizontal*, o arquivo é distribuído entre todos os servidores de dados disponíveis. Este layout de distribuição explora o paralelismo máximo oferecido pelo sistema. Em contrapartida, no *layout vertical*, cada arquivo é armazenado integralmente em um único servidor de dados. Neste layout, operações sobre um mesmo arquivo não são paralelizáveis. Contudo, se uma distribuição uniforme dos arquivos for realizada,

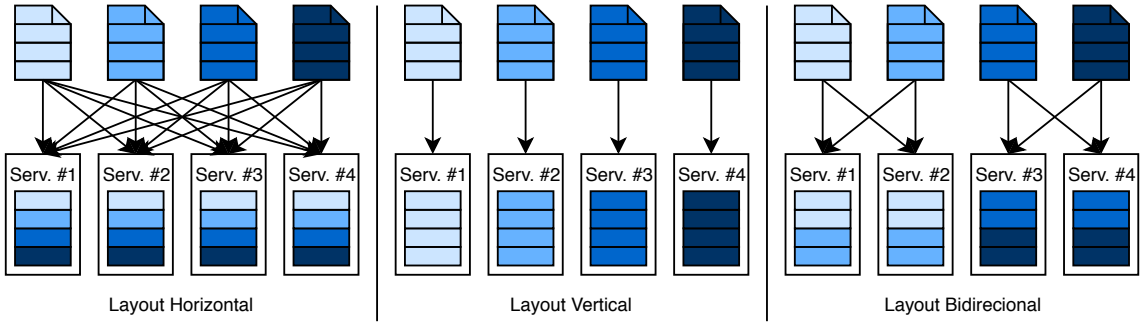


Figura 1.4. Layouts de distribuição de arquivos em SAPs (Adaptado de [Song et al. 2012]).

como no exemplo, o layout vertical pode reduzir a contenção provocada por diferentes processos acessando um mesmo servidor de dados. O *layout bidirecional* apresenta-se como um compromisso entre os layouts horizontal e vertical. No layout bidirecional, o arquivo é distribuído ao longo de uma fração dos servidores de dados, experimentando assim algum nível de paralelismo, ao mesmo tempo que a contenção entre processos em um mesmo servidor de dados é reduzida.

1.3. E/S em Aplicações Distribuídas

Quase que invariavelmente, uma aplicação distribuída de larga escala irá, em algum momento da sua execução, ler ou escrever dados em arquivos do sistema de armazenamento secundário. Isso significa que a aplicação deverá realizar operações, pelo menos, para *criar* ou *abrir* um arquivo, *ler* ou *escrever* uma quantidade de dados neste arquivo, e, finalmente, *fechar* o arquivo. Estas são as operações mais básicas, e comumente encontradas nas aplicações, com relação a E/S de dados em arquivos. Nesta seção serão discutidas diferentes maneiras de se realizar estas operações, considerando algumas das principais APIs utilizadas na área.

Para guiar essa discussão, será utilizado um estudo de caso baseado em um problema real. Considere um conjunto de dados Cartesiano, como o apresentado na Figura 1.5. Este conjunto de dados corresponde a um intervalo de tempo simulado pelo modelo NICAM, utilizado para o estudo de convecção atmosférica global de alta umidade em escala sub-quilométrica [Miyamoto et al. 2013]. As dimensões deste conjunto de dados tridimensional são de 11.520 pontos no eixo x, 5.760 pontos no eixo y e 94 pontos no eixo z. No conjunto de dados real, múltiplas variáveis são computadas para cada ponto. Neste estudo de caso, para fins de simplificação, consideraremos uma única variável de ponto flutuante com precisão simples (*float*), de 4 bytes, por ponto. Como resultado, tem-se um conjunto de dados com um tamanho total de aproximadamente 23 GiB (24.949.555.200 bytes). Adicionalmente, considere neste estudo de caso que o conjunto de dados será armazenado em um único arquivo.

A simulação na qual este conjunto de dados foi baseado, utilizou-se de todos os 82.944 nodos de computação do supercomputador japonês *K computer*, para simular 48 intervalos de tempo. Estes números mostram a necessidade de dividir o domínio do problema entre múltiplos processos para que a computação se torne praticável. Idealmente,

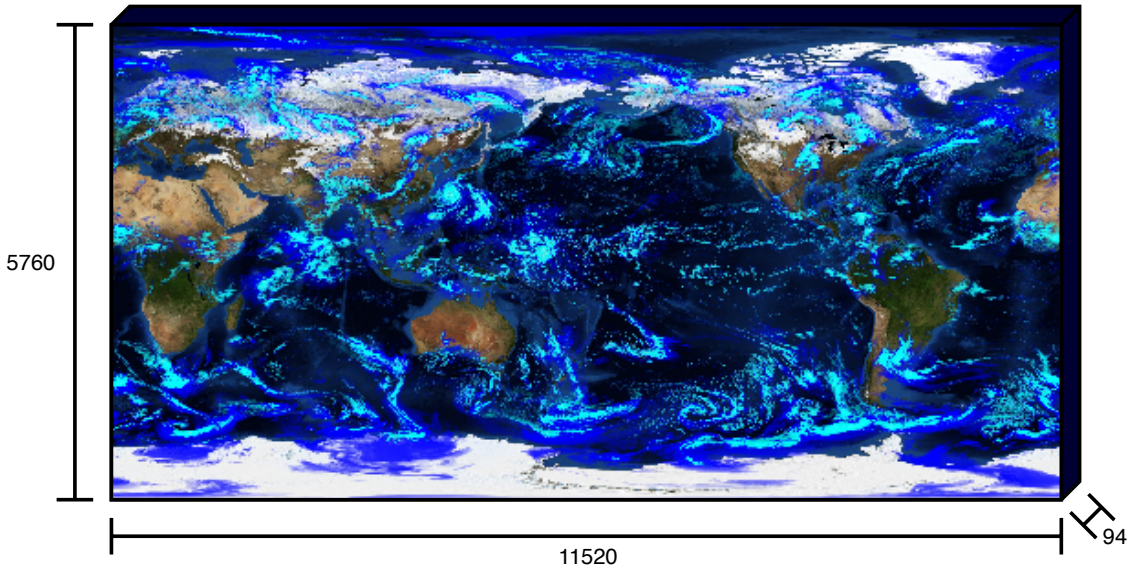


Figura 1.5. Exemplo de um conjunto de dados Cartesiano real extraído da saída do modelo de simulação NICAM [Miyamoto et al. 2013]. Este conjunto de dados apresenta 11.520 pontos no eixo x, 5.760 pontos no eixo y e 94 pontos no eixo z.

o domínio do problema é dividido igualmente entre os processos disponíveis, o que, por consequência, se traduziria em cada processo lendo ou escrevendo o mesmo volume de dados. Mesmo nestas condições, diferentes maneiras de se dividir o conjunto de dados entre os múltiplos processos podem resultar em variações significativas no desempenho de E/S da aplicação [Inacio et al. 2017b]. Neste estudo de caso, em favor da simplicidade, o domínio do problema será dividido entre 40 processos, sendo que cada um será responsável por uma seção cúbica do conjunto de dados com 1.152 pontos no eixo x, 1.440 pontos no eixo y e 94 pontos no eixo z, totalizando aproximadamente 595 MiB (623.738.880 bytes) de dados por processo.

1.3.1. POSIX

A especificação PORTABLE OPERATING SYSTEM INTERFACE (POSIX) [IEEE and The Open Group 2013] define funções padrão para sistemas operacionais, focando, principalmente, na portabilidade de software. Entre as funções definidas pelo padrão POSIX, existe um grande número voltado para permitir o acesso a dados em arquivos. Estas funções podem ser agrupadas de acordo com o mecanismo utilizado para representar a conexão entre a aplicação e o arquivo: descritores de arquivos e *stream*.

Funções baseadas em descritores de arquivos oferecem uma interface mais primitiva e de mais baixo nível para as aplicações. Por outro lado, funções baseadas em *stream* apresentam facilidades diferenciadas, como *buffering*, e são usualmente implementadas sobre funções mais básicas baseadas em descritores de arquivos. Além de especificadas na POSIX, as funções de E/S baseadas em *stream* são características do padrão da linguagem de programação C. Esta seção focará nas chamadas de sistema baseadas em descritores de arquivos da POSIX, enquanto a seção seguinte abordará funções de E/S do

padrão C baseadas em *stream*.

Primeiramente, para ter acesso a um arquivo no sistema de arquivos, a aplicação precisa estabelecer uma conexão com o mesmo. Para isso, utiliza-se a função `open()`. Esta função recebe como argumentos o caminho para o arquivo, que pode ser relativo ou absoluto, e *flags*, que indicam o estado e o modo de acesso do arquivo, como a *flag* `O_CREAT`, que define que, se o arquivo informado não existir no sistema de arquivos, este deve ser criado quando da chamada da função. O retorno da função `open()` é um número inteiro que identifica o descritor do arquivo. No estudo de caso em questão, cada um dos 40 processos precisa estabelecer a conexão com o arquivo antes de acessar sua porção do conjunto de dados.

Para ler e escrever dados no arquivo, a aplicação pode utilizar, respectivamente, as funções `read()` e `write()` da POSIX. Ambas recebem como argumentos o descritor do arquivo, um ponteiro para um endereço de memória contendo os dados para serem escritos ou alocado para receber os dados lidos, e o número de bytes a serem transferidos. Um detalhe particular que surge no contexto de aplicações distribuídas cujos processos acessam dados em um arquivo compartilhado, como no estudo de caso apresentado, é a definição de em qual região do arquivo estão os dados desejados.

Por padrão, a função `open()` define o marcador com a posição atual no arquivo para o início do arquivo. Portanto, se cada processo, por exemplo, escrever sua porção do conjunto de dados no arquivo compartilhado sem antes alterar este marcador, os dados serão sobrescritos a cada operação realizada pelos processos. Para redefinir este marcador, a POSIX oferece a função `lseek()`, que recebe como argumentos o descritor do arquivo, um *offset* (em bytes) e a posição a partir da qual o offset deve ser contabilizado. Vale ressaltar que o offset do arquivo é incrementado automaticamente toda vez que as funções `read()` e `write()` são invocadas, utilizando o número de bytes transferidos. Alternativamente, para evitar a necessidade de invocar as funções `lseek()` antes das leituras e escritas, a POSIX oferece as funções `pread()` e `pwrite()`, que recebem um argumento adicional com o offset a partir do início do arquivo onde a operação deve ser realizada e não movem o marcador do offset do arquivo ao final da execução. Esta facilidade é particularmente interessante quando a aplicação tem por característica realizar acessos a regiões não-contíguas do arquivo.

Independente da utilização das funções `lseek()` e `read()`/`write()` ou das funções `pread()`/`pwrite()`, o desenvolvedor da aplicação precisa garantir que cada processo acessará os offsets corretos no arquivo compartilhado, o que pode ser considerado uma atividade não trivial e propensa a erros. Primeiramente, é preciso considerar como os dados são armazenados no arquivo. Considerando este estudo de caso, por exemplo, onde o conjunto de dados é estruturado como um espaço Cartesiano tridimensional, uma convenção típica em aplicações escritas na linguagem C consiste em ordenar estes dados no arquivo por linhas (*row-major order*). No estudo de caso, isso se traduz em armazenar, para cada ponto do eixo x, todos os pontos do eixo y e, para cada ponto do eixo y, todos os pontos do eixo z, conforme ilustrado na Figura 1.6. Desta forma, observa-se que o conhecimento sobre a estrutura do domínio do problema é fundamental para a coordenação do acesso distribuído e concorrente aos dados. Em posse de um referencial do processo dentro da aplicação distribuída, como o *rank* MPI, por exemplo, é possí-

vel se definir uma fórmula para se calcular os offsets para cada operação de E/S de cada processo.

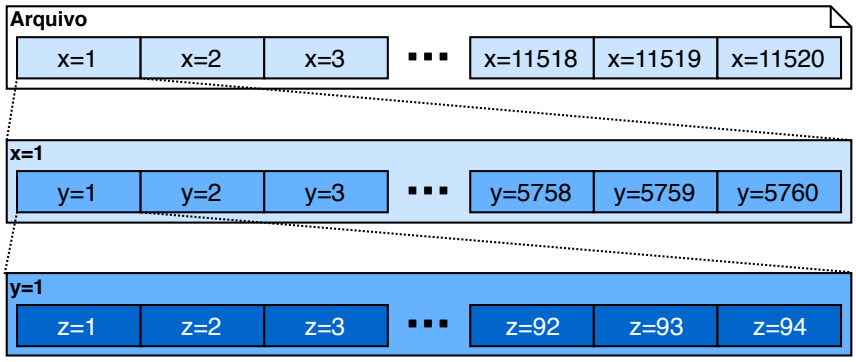


Figura 1.6. Serialização do conjunto de dados Cartesiano do estudo de caso em um arquivo.

Finalmente, uma vez transferidos todos os dados de ou para o arquivo, a aplicação fecha a conexão com o arquivo utilizando a chamada de sistema POSIX `close()`, passando como único argumento o descritor do arquivo. Assim como no estabelecimento da conexão, todos os 40 processos, considerando este estudo de caso, devem fechar a conexão com o arquivo ao final da sua utilização. Esta é uma boa prática que visa evitar diversos problemas, entre eles, o bloqueio de arquivos.

1.3.2. Padrão C (Stream de E/S)

Como mencionado anteriormente, o padrão da linguagem de programação C define algumas funções de alto nível para E/S em arquivos baseadas em *stream* [Free Software Foundation 2018]. Estas funções são usualmente implementadas sobre funções de mais baixo nível, como as chamadas de sistema POSIX baseadas em descritores de arquivos apresentadas na seção anterior. A interface baseada em *stream* apresenta algumas vantagens, principalmente para operações de E/S que realizam transferência de dados.

Além da oferta de funções mais elaboradas, como o suporte para formatação de E/S, por exemplo, as funções baseadas em *stream* utilizam um mecanismo de *buffering* para transferência de caracteres. Durante operações de escrita, os caracteres são acumulados e transferidos assincronamente para o arquivo em blocos, ao invés de aparecer imediatamente após executada a função. De maneira similar, em operações de leitura, os dados do arquivo são recuperados em blocos e não caractere por caractere [Free Software Foundation 2018]. Esta facilidade visa, principalmente, minimizar o impacto do acesso ao sistema de armazenamento secundário, cujo desempenho é consideravelmente inferior ao das memórias principais.

A função `fopen()` estabelece a conexão com um arquivo, cujo caminho é passado como primeiro argumento, e retorna o *stream* associado com esta conexão. O segundo argumento desta função corresponde ao modo de acesso. Dependendo do modo de acesso, que inclui somente leitura, somente escrita, leitura e escrita e acréscimo ao final do arquivo (*append*), o marcador com a posição do arquivo pode variar entre o início e o final do arquivo. Sendo assim, como foi observado na seção anterior, é necessário contro-

lar o marcador da posição do arquivo de forma que os diferentes processos da aplicação distribuída acessem a sua região específica do conjunto de dados adequadamente. Para isso, o padrão C disponibiliza as funções `fseek()` e `rewind()`. A função `fseek()` é equivalente a chamada de sistema POSIX `lseek()`, com a exceção de que o primeiro argumento é um *stream* e não um descritor de arquivo. A função `rewind()` é um forma simplificada da `fseek()` que recebe como argumento apenas o *stream* e redefine o marcador da posição do arquivo para o seu início.

As funções `fread()` e `fwrite()` possibilitam, respectivamente, ler e escrever dados em um arquivo. Ambas as funções recebem como argumentos um endereço de memória apontando para onde estão os dados a serem escritos ou para onde os dados a serem lidos devem ser colocados; o tamanho (em bytes) de cada item a ser lido ou escrito, por exemplo, um byte no caso de um caractere; o número de itens; e o *stream* do arquivo. Ao final da execução, ambas as funções movem o marcador da posição no arquivo de acordo com a quantidade de dados acessada. Não há no padrão C funções equivalentes às chamadas de sistema POSIX `pwrite()` e `pread()`, que permitem passar a posição no arquivo onde a operação deve ser realizada como argumento. Portanto, para estabelecer o padrão de acesso não-contíguo, como o demandado pelo estudo de caso apresentado, é necessário realizar duas operações: uma de posicionamento no arquivo, com `fseek()`, e outra para acessar os dados, com `fread()` ou `fwrite()`.

Assim como discutido anteriormente, ao final do acesso aos dados do arquivo, a aplicação deve encerrar a conexão com o mesmo. A função `fclose()` é responsável por concluir a transferência de todos os dados acumulados no *buffer* para o arquivo e fechar a conexão com o *stream* passado como único argumento. A transferência dos dados acumulados no *buffer* pode ser realizada antes do fechamento do arquivo, utilizando a função `fflush()`, que recebe o *stream* do arquivo como argumento. Esta função força a aplicação a aguardar até que todos os dados em *buffer* sejam transferidos para o armazenamento secundário, o que pode resultar em uma degradação de desempenho.

1.3.3. MPI-IO

MPI-IO é uma especificação de interface para E/S paralela em arquivos definida a partir da segunda versão do padrão MESSAGE PASSING INTERFACE (MPI) [MPI Forum 1997]. Esta especificação define um amplo conjunto de funções e facilidades projetadas para prover maior eficiência nas operações de E/S realizadas por aplicações distribuídas, principalmente, àquelas baseadas em MPI. Localizada normalmente numa camada intermediária, entre o SAP e a aplicação, MPI-IO é frequentemente referido como um *middleware* de E/S [Prabhat and Koziol 2014].

MPI-IO suporta os três métodos fundamentais de se realizar E/S em arquivos em uma aplicação distribuída [Prabhat and Koziol 2014], ilustradas na Figura 1.7. No primeiro deles, conhecido como *arquivo por processo*, cada processo escreve em um arquivo independente. Apesar de ser um método consideravelmente simples e de oferecer alto paralelismo, apresenta como desvantagem o grande número de potencialmente pequenos arquivos. O segundo método visa resolver este problema, *concentrando* todos os dados em um único processo, que os transfere para um único arquivo. Evidentemente, além de não explorar nenhum tipo de paralelismo, esta não é uma abordagem escalável.

Congestionamentos de comunicação podem ter um impacto negativo no desempenho das aplicações devido ao grande número de processos, assim como a capacidade de memória em um único processo poder não ser suficiente para acumular os dados enviados pelos demais. O terceiro método, particularmente explorado no MPI-IO, busca prover uma solução para as questões anteriores. Neste método, todos os processos acessam dados em um *arquivo compartilhado* de maneira coordenada. O restante desta seção se dedicará a prover maiores detalhes sobre este método no MPI-IO.

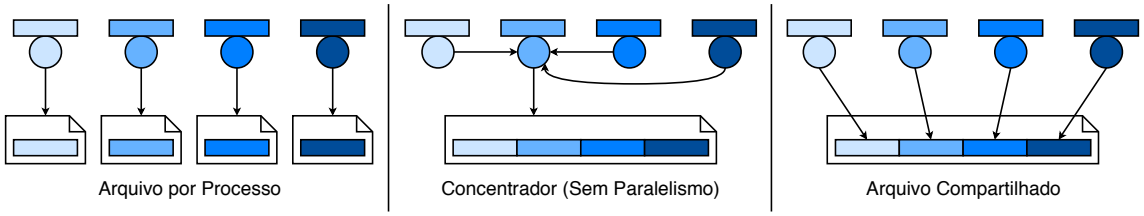


Figura 1.7. Três métodos fundamentais para se realizar E/S em uma aplicação distribuída.

Existem dois grupos de funções de E/S no MPI-IO: independentes e coletivas [MPI Forum 1997]. Apesar de serem muito similares na definição, as funções pertencentes a cada um destes dois grupos têm uma diferença fundamental. As funções de E/S *independentes* podem ser executadas por um processo sem depender dos demais processos da aplicação distribuída, similar ao que ocorre com as chamadas de sistema POSIX e com a funções de *stream* de E/S do padrão C. Por outro lado, as funções de E/S *coletivas* requerem que todos os processos vinculados a um comunicador MPI, executem a mesma função para que a aplicação possa prosseguir.

1.3.3.1. E/S Independente

Como mencionado anteriormente, a E/S independente no MPI-IO se assemelha na estrutura e aparência à E/S POSIX e do padrão C. Para mover o marcador de posição no arquivo, utiliza-se a função `MPI_File_seek()`, que recebe como argumentos a referência para o arquivo aberto, um offset (em bytes) e a posição a partir da qual o offset deve ser contabilizado. Vale ressaltar que esta função move apenas o marcador na referência do arquivo para o processo invocador, sem afetar os demais processos. As funções `MPI_File_read()` e `MPI_File_write()` realizam, respectivamente, a leitura e escrita de dados em um arquivo. Os argumentos para ambas as funções são: a referência para o arquivo aberto, um endereço de memória apontando para ou de onde os dados devem ser transferidos, o quantidade de elementos na memória, o tipo do elemento (caractere, inteiro, tipo customizado, etc.) e um objeto `MPI_Status` (que pode ser ignorado passando `MPI_STATUS_IGNORE`), que armazena dados referentes a comunicação realizada durante a execução da função.

MPI-IO oferece alternativas para funções de leitura e escrita em que o offset do arquivo onde a operação deve ser executada é passado como argumento adicional: `MPI_File_read_at()` e `MPI_File_write_at()`. Estas funções evitam a realização de duas chamadas de funções para se realizar uma leitura e escrita, uma para mover

a posição no arquivo e outra para acessar os dados. Porém, vale lembrar que o marcador da posição do arquivo não é atualizado ao final da execução destas funções.

As funções para abertura (criação) e fechamento de arquivos no MPI-IO, obrigatórias antes e depois, respectivamente, do acesso aos dados de um arquivo, são *coletivas* sobre o comunicador MPI. Isto significa que todos os processos vinculados ao comunicador MPI passado como argumento para as funções de abertura (criação) e fechamento de um arquivo, precisam necessariamente invocar a referida função para que a aplicação possa prosseguir. Por esta razão, estas funções serão discutidas na próxima seção, junto com as demais funções de E/S coletivas do MPI-IO.

1.3.3.2. E/S Coletiva

Para estabelecer uma conexão com um arquivo, o MPI-IO disponibiliza a função coletiva `MPI_File_open()`. Esta função recebe como argumentos um comunicador MPI, que estabelece os processos participantes da operação coletiva; o caminho do arquivo; o modo de abertura, que permite definir, entre outras opções, que um arquivo seja criado caso este não exista; um objeto `MPI_Info`, que permite passar informações adicionais para o MPI-IO, possibilitando otimizações mais específicas; e um endereço de memória para um objeto `MPI_File`, onde será armazenada a referência do arquivo aberto. O objeto `MPI_File` é utilizado pelas demais funções de E/S do MPI-IO, sejam elas coletivas ou independentes, de maneira similar ao descritor de arquivo no POSIX ou o *stream* nas funções de E/S do padrão C.

As funções de leitura e escrita coletivas do MPI-IO são idênticas em termos da lista de argumentos às suas respectivas funções independentes. Os nomes destas funções coletivas apresentam uma pequena diferença, com o acréscimo do sufixo `_all`: `MPI_File_read_all()`,`MPI_File_write_all()`,`MPI_File_read_at_all()` e `MPI_File_write_at_all()`. Estas funções, quando invocadas dentro da aplicação distribuída, precisam ser feitas por todos os processos vinculados ao comunicador MPI da abertura do arquivo para que a operação seja realizada e a aplicação possa prosseguir sua execução.

Mesmo tratando-se de funções coletivas, vale notar que cada processo pode acessar uma região diferente do arquivo. Basta utilizar a função `MPI_File_seek()` para mover o marcador da posição no arquivo antes das funções `MPI_File_read_all()` e `MPI_File_write_all()`, ou utilizando diretamente as funções que recebem o offset como argumento `MPI_File_read_at_all()` e `MPI_File_write_at_all()`. Esta é uma característica de particular interesse, por exemplo, para o estudo de caso apresentado, onde cada um dos 40 processos acessa determinados pontos dentro do conjunto de dados armazenados em offsets específicos do arquivo compartilhado.

A função `MPI_File_close()`, que recebe como argumento único a referência para o arquivo (objeto `MPI_File`), fecha a conexão com o arquivo. Esta função, assim como as demais funções coletivas mencionadas anteriormente, precisa ser chamada por todos os processos vinculados ao comunicador MPI utilizado no momento da abertura do arquivo compartilhado. Percebe-se que, independente da complexidade e das facilidades da API utilizada, o fluxo para acesso aos dados é o mesmo. O arquivo deve ser aberto

antes de ser acessado, seja para leitura ou escrita, e fechado ao final do acesso.

As funções de leitura e escrita do MPI-IO, sejam elas independentes ou coletivas, apresentadas até este ponto foram discutidas em termos de offsets específicos. Entre outras palavras, considerando múltiplos processos acessando um arquivo compartilhado, como no estudo de caso apresentado, cada processo precisa mover o marcador para a posição adequada no arquivo antes de fazer a leitura ou escrita, ou utilizar uma função que aceite o offset no arquivo como argumento. Esta abordagem de relativo baixo nível, suportada por outras APIs já discutidas como POSIX e do padrão C, é bastante flexível, uma vez que permite controle total sobre a região do arquivo a ser acessada. Por outro lado, em cenários como o do estudo de caso apresentado, em que um conjunto de dados Cartesiano tridimensional é dividido em subconjuntos, também tridimensionais, para múltiplos processos, que precisam acessá-lo em um arquivo compartilhado, a representação das regiões visíveis para cada processo através de um conjunto de offsets não é natural e pode ser tornar uma tarefa desafiadora e propensa a erros. Visando endereçar esta questão, o MPI-IO oferece um mecanismo para possibilitar uma representação mais natural de acessos não-contíguos em um arquivo.

1.3.3.3. Acesso Não-contíguo

Duas facilidades particulares do MPI e do MPI-IO favorecem a programação de acesso a dados não-contíguos em arquivos: tipos de dados customizados e visão de arquivo. Todas as funções de leitura e escrita do MPI-IO descritas nas seções anteriores realizam a transferência de dados baseando-se no tipo do elemento (dado) e na quantidade de elementos. O MPI-IO oferece um conjunto de tipos de dados básicos, como caracteres, inteiros, número de ponto flutuante, entre outros, para descrever estes elementos. Porém, em algumas situações, o elemento pode ser de um tipo mais complexo, composto por vários tipos básicos, como um `struct` do C, por exemplo. Para possibilitar a descrição deste tipo de elemento, o MPI-IO oferece um conjunto de funções para definição de *tipos de dados customizados*. A função `MPI_Type_create_struct()`, por exemplo, permite a definição de um tipo de dado customizado para representar uma `struct` que a aplicação pretende armazenar ou recuperar de um arquivo utilizando uma única operação de escrita ou leitura. Um detalhe importante é que, uma vez criado o tipo, este precisa ser registrado com a função `MPI_Type_commit()` antes de poder ser utilizado pela aplicação.

A *visão de arquivo* é uma facilidade que visa possibilitar a definição, de maneira mais natural, de quais regiões do arquivo compartilhado são visíveis para cada processo. Isto inclui a possibilidade de definir várias regiões não-contíguas no arquivo. A visão de um arquivo é definida após a sua abertura, utilizando a função `MPI_File_set_view()`. Esta função recebe como argumentos a referência do arquivo aberto, um deslocamento (em bytes), contado a partir do início do arquivo; o tipo de dados elementar, correspondente ao menor elemento que compõe a região, que pode ser tanto um tipo básico quando um tipo customizado do MPI; um tipo de dados representando o arquivo, que estabelece quais regiões do arquivo são visíveis para o processo e que deve ser o mesmo tipo de dados elementar ou derivado deste; o tipo de representação de dados; e um objeto `MPI_Info` com informações adicionais para auxiliar em otimizações mais específicas.

Uma vez definida a visão do arquivo para o processo, este poderia acessar, utilizando uma única chamada para uma função de leitura e escrita, toda a sua parte do conjunto de dados, mesmo que esta seja composta por regiões não-contíguas.

A visão do arquivo provê uma interpretação global do tipo de acesso que os processos de uma aplicação distribuída podem realizar em um arquivo compartilhado. Esta interpretação, combinada à coordenação oferecida pelas funções coletivas do MPI-IO, permite que diferentes otimizações sejam empregadas de forma que operações de leitura e escrita sejam realizadas de maneira mais eficiente e, por consequência, em um menor intervalo de tempo. Entre as otimizações mais populares estão a *data sieving* e a *two-phase I/O*. A técnica de *data sieving* [Thakur et al. 2002] consiste de acessar uma região do arquivo maior do que a requisitada pelo processo e, em memória, extrair os dados de fato solicitados. Esta técnica visa evitar a realização de um grande número de acessos a pequenas regiões de um arquivo, um padrão de acesso reconhecidamente ruim do ponto de vista de desempenho de E/S em SAPs. Na técnica de *two-phase I/O* [Thakur et al. 1999], operações coletivas de leitura ou escrita são divididas em duas fases, uma de agregação e outra de acesso ao arquivo, conforme ilustrado na Figura 1.8 para o estudo de caso apresentado.

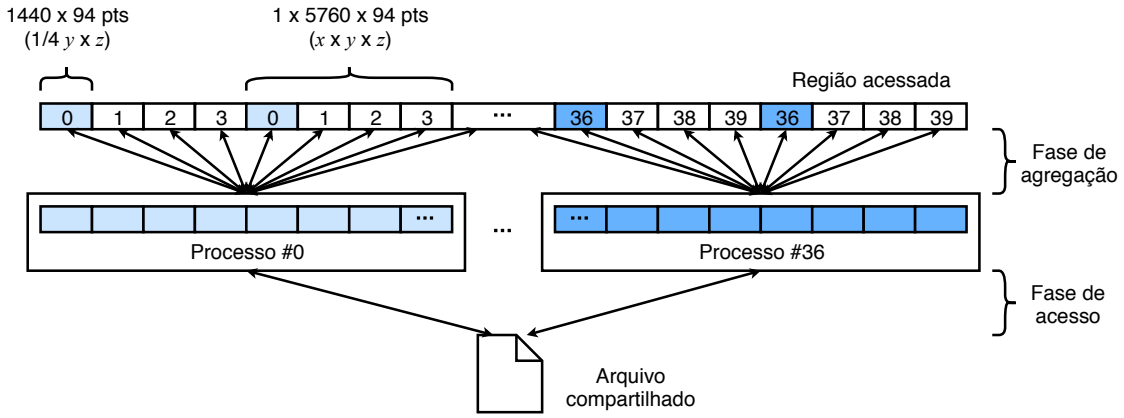


Figura 1.8. Exemplo da técnica de otimização *two-phase I/O* aplicada a uma operação de E/S coletiva do MPI-IO no conjunto de dados do estudo de caso.

Considere, por exemplo, uma operação de escrita coletiva, onde uma visão do arquivo é definida envolvendo os 40 processos do estudo de caso, cada um enviando para um arquivo compartilhado a sua porção tridimensional do conjunto de dados. Na *fase de agregação*, um subconjunto dos processos participantes atuam como agregadores. No exemplo da Figura 1.8, um a cada quatro processos são agregadores (ex.: processos 0 e 36). Cada agregador recebe de outros processos dados pertencentes a outras regiões adjacentes ou próximas a própria região acessada pelo agregador. Uma vez tendo agregado os dados de outros processos, formando uma região contígua maior que a região que seria acessada individualmente por cada processo, inicia-se a *fase de acesso*, em que os agregadores realizam a transferência dos dados agregados para o arquivo compartilhado. Numa leitura coletiva, o processo basicamente se inverte, ocorrendo primeiro a fase de acesso e depois a fase de distribuição (em oposição a fase de agregação), em que os dados seriam

encaminhados para cada processo solicitante de acordo com a região acessada por estes.

1.4. Ferramentas de Avaliação de Desempenho

Embora diferentes APIs e funções de E/S, das mais simples às mais sofisticadas, estejam a disposição, encontrar a alternativa mais eficiente em termos de desempenho de acesso a grandes conjuntos de dados continua sendo um desafio. Isto porque o desempenho de E/S em aplicações distribuídas de larga escala que fazem uso intensivo de dados depende de um grande número de fatores, incluindo aspectos tanto da infraestrutura de armazenamento quanto da pilha de software de E/S [Carns et al. 2009, Lofstead et al. 2010, Inacio et al. 2015a, Inacio et al. 2015b, Herbein et al. 2016, Inacio et al. 2017a]. Muitas vezes, diferentes cenários, construídos a partir da combinação de diferentes métodos de E/S e parâmetros do sistema de armazenamento, são avaliados através de experimentos para se poder identificar qual destes se mostra mais eficiente [Boito et al. 2018]. Para auxiliar neste processo, diversas ferramentas especializadas na reprodução de padrões de acesso a dados em arquivos e avaliação de desempenho de E/S foram desenvolvidas. Nesta seção, serão apresentadas algumas das ferramentas mais populares na comunidade de pesquisa em E/S paralela.

1.4.1. mpi-tile-io

MPI-TILE-IO [ANL 2002] é uma ferramenta para avaliação de desempenho de E/S paralela em acessos não-contíguos utilizando funções de E/S do MPI-IO. O MPI-TILE-IO reproduz um conjunto específico de dados: uma matriz densa bidimensional. Cada processo participante é responsável por uma parte, denominada *tile*, desta matriz e a escreve em um arquivo compartilhado por meio de funções independentes ou coletivas do MPI-IO. Este tipo de carga de trabalho é bastante comum entre aplicações de visualização e de simulação científicas e de engenharia.

Embora o MPI-TILE-IO gere dados apenas segundo um layout de matriz bidimensional, uma lista de parâmetros é oferecida para customização do conjunto de dados gerado, como demonstrado na Figura 1.9:

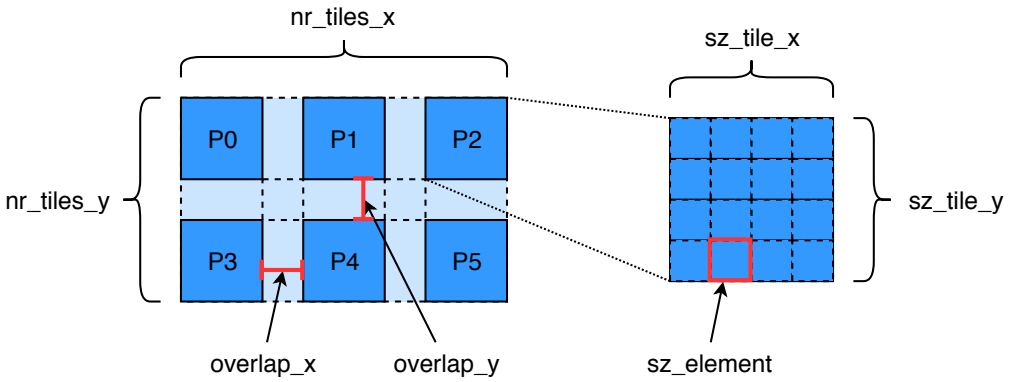


Figura 1.9. Exemplo de um conjunto de dados gerado pelo MPI-TILE-IO, destacando seus parâmetros de execução.

- *nr_tiles_x*: número de *tiles* no eixo *x* da matriz;

- *nr_tiles_y*: número de *tiles* no eixo *y* da matriz;
- *sz_tile_x*: número de elementos no eixo *x* de cada *tile*;
- *sz_tile_y*: número de elementos no eixo *y* de cada *tile*;
- *sz_element*: tamanho de um elemento em bytes;
- *overlap_x*: número de elementos compartilhados entre dois *tiles* adjacentes no sentido do eixo *x*;
- *overlap_y*: número de elementos compartilhados entre dois *tiles* adjacentes no sentido do eixo *y*.

A simplicidade do projeto e desenvolvimento do MPI-TILE-IO, combinada a representatividade da carga de trabalho gerada, contribuíram para sua utilização na comunidade de pesquisa em E/S paralela. Contudo, a sua especificidade limita sua utilização em estudos mais amplos, para investigação de diferentes cargas de trabalho e métodos de E/S. Por esta razão, muitos trabalhos de pesquisa se utilizam de outros benchmarks de E/S em conjunto com o MPI-TILE-IO. Visando concentrar esta demanda de flexibilidade, outras ferramentas foram propostas.

1.4.2. Interleaved-or-Random (IOR)

O INTERLEAVED-OR-RANDOM (IOR) [LLNL 2013, Shan et al. 2008] é um benchmark de E/S paralela que permite reproduzir um amplo e variado conjunto de cargas de trabalho. Por meio de parâmetros relativamente simples, o IOR permite avaliar o desempenho de diferentes APIs de E/S, como POSIX, MPI-IO, HDF5 e PNETCDF. Estes parâmetros permitem também controlar características das cargas de trabalho geradas, como acessos a dados contíguos e não-contíguos, em arquivos compartilhados ou individuais, quantidade de dados e tamanho de requisições, entre outros. Esta flexibilidade oferecida pelo IOR a transformou em um padrão "de facto" na comunidade de pesquisa em E/S paralela, sendo uma das ferramentas mais utilizadas para avaliação de desempenho de E/S paralela na área de CAD [Boito et al. 2018].

Na versão 3.0.1 do IOR, cerca de 50 parâmetros são disponibilizados. A Figura 1.10 apresenta três dos principais parâmetros, utilizados para a definição da carga de trabalho gerada por cada processo do IOR. O parâmetro *transferSize* estabelece o tamanho (em bytes) de cada operação de E/S realizada, seja ela leitura ou escrita. O parâmetro *blockSize* define o tamanho (em bytes) de um bloco de dados a ser acessado. Na prática, este parâmetro estabelece o número de operações de E/S que serão realizadas, uma vez que seu valor deve ser um múltiplo do parâmetro *transferSize*. Em um cenário de acesso sequencial, como o do exemplo da Figura 1.10, cada processo acessa uma região contígua de tamanho *blockSize*, realizando operações de E/S de tamanho *transferSize*. A região contígua que compreende os blocos acessados por todos os processos é denominada de *segmento*. O parâmetro *segmentCount* define quantos segmentos existem no arquivo. No caso do exemplo da Figura 1.10, há dois segmentos. Os segmentos permitem representar padrões de acesso não-contíguos, onde cada processo acessa blocos de dados separados por intervalos regulares.

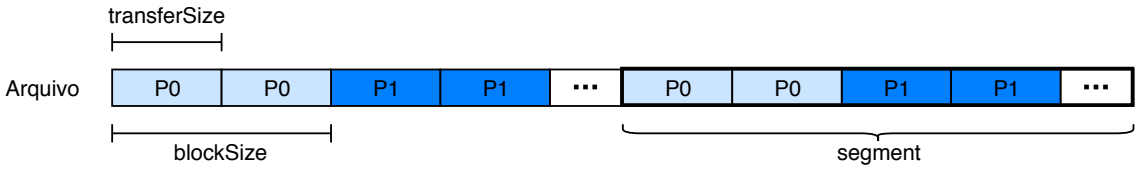


Figura 1.10. Estrutura de um arquivo compartilhado por múltiplos processos gerado pelo IOR, destacando alguns dos parâmetros oferecidos para definição da carga de trabalho.

Apesar das inúmeras facilidades providas pelo IOR, este apresenta algumas limitações. Como mencionado, as cargas de trabalho geradas pelo IOR são obrigatoriamente homogêneas, no sentido que todos os processos acessam a mesma quantidade de dados, utilizando operações também de mesmo tamanho. Além disso, a facilidade de se definir segmentos, oferecida pelo IOR, não é suficiente para se representar precisamente algumas cargas de trabalho usualmente encontradas em aplicações de CAD, como, por exemplo, algumas matrizes bidimensionais, como as geradas pelo MPI-TILE-IO. Estas e outras limitações, aliadas a descontinuidade no desenvolvimento do IOR, motivaram a proposta de uma ferramenta alternativa.

1.4.3. IOR-Extended (IORE)

O IOR-EXTENDED (IORE) [Inacio and Dantas 2018b, Inacio and Dantas 2018a] foi originalmente projetado como uma extensão do benchmark IOR. O foco inicial era endereçar algumas limitações particulares do IOR, como a homogeneidade da carga de trabalho, por exemplo. Porém, mais do que um gerador de carga de trabalho, o IORE evoluiu para uma ferramenta diferenciada para avaliação de desempenho de E/S paralela e de sistemas de armazenamento distribuído. Um conjunto de novas funcionalidades foi incluído no projeto do IORE visando não apenas ampliar e aprimorar aspectos relacionados a geração de carga de trabalho, mas também agilizar a realização de experimentos e análises de desempenho, e aumentar a reprodutibilidade dos resultados.

Uma das principais funcionalidades introduzidas no IORE é a *execução orientada por experimento*. Basicamente, toda a execução do IORE consiste de um *experimento*, cuja estrutura é apresentada na Figura 1.11. Cada experimento consiste de uma ou mais *rodadas*, que, por sua vez, consiste de um conjunto de parâmetros de configuração que define as características de um teste. Rodadas podem ser *repetidas* consecutivamente, assim como experimentos podem ser *replicados* como um todo, mantendo a ordem original de rodadas ou executando as rodadas em ordem aleatória. Esta estrutura provê flexibilidade para o usuário, que pode utilizar o IORE tanto para avaliações de desempenho rápidas, quanto para estudos mais complexos, envolvendo a análise de múltiplas variáveis.

Outro aprimoramento introduzido no IORE é o suporte para definição de *cargas de trabalho baseadas em conjuntos de dados*. Os parâmetros oferecidos pelo IOR, como apresentado na seção anterior, permitem definir a carga de trabalho gerada sob uma perspectiva de volume de dados e offsets. Embora flexível, pode-se observar que determinados conjuntos de dados típicos de aplicações de CAD não são precisamente representadas utilizando-se tais parâmetros. Através de parâmetros simples, porém específicos por tipo

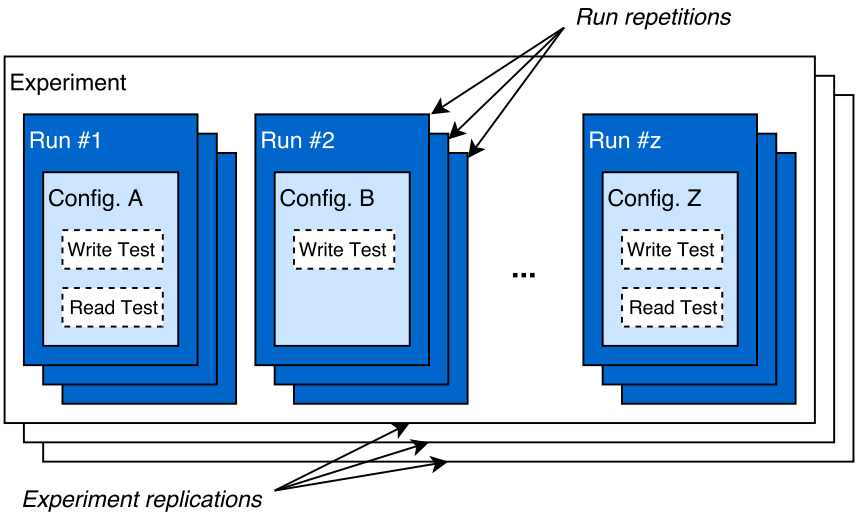
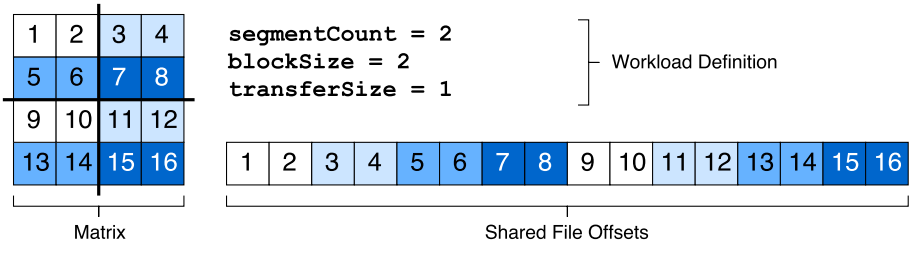
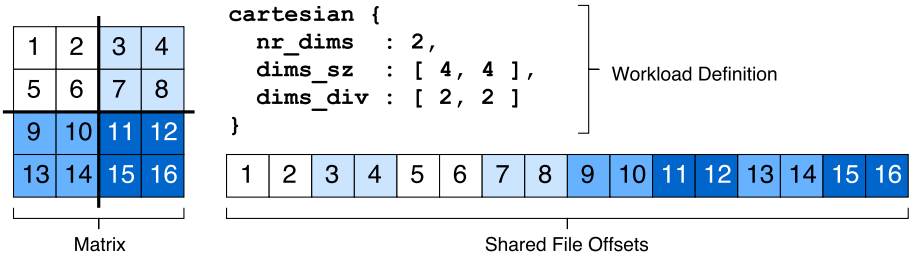


Figura 1.11. Estrutura de um experimento no IORE.

de conjunto de dados, o IORE permite que tais cargas de trabalho possam ser geradas, conforme ilustrado no exemplo da Figura 1.12. Neste exemplo, a proposta é que uma matriz bidimensional de dimensões 4x4 seja dividida entre quatro processos, de forma que cada um atue sobre uma submatriz 2x2. Na Figura 1.12a, observa-se que, utilizando os parâmetros oferecidos pelo IOR, a carga de trabalho não reflete exatamente a proposta. Apesar de cada processo acessar o volume de dados correto, os offsets acessados não são os esperados. O modelo de definição de conjunto de dados Cartesianos oferecido pelo IORE, como demonstrado na Figura 1.12b, permite representar precisamente a carga de trabalho esperada para este exemplo.



(a) IOR



(b) IORE

Figura 1.12. Representação de uma matriz bidimensional usando o IOR e o IORE.

Além de suportar a geração de cargas de trabalho homogêneas, assim como no IOR, o IORE oferece também suporte para cargas de trabalho heterogêneas. Listas de quantidades de dados e tamanho de requisições podem ser passadas como parâmetro, assim como distribuições de probabilidade (ex.: normal, uniforme, geométrica), de forma que cada processo possa gerar uma carga de trabalho diferente dos demais. O IORE oferece ainda parâmetros para manipulação de opções de configuração do sistema de arquivos paralelo, como tamanho de faixa, por exemplo, em tempo de execução; e exportação das medidas de desempenho coletadas em arquivo CSV ao final da execução, evitando a necessidade de tratamento de informações enviadas para saída padrão.

O projeto e desenvolvimento do IORE faz parte de um trabalho de pesquisa em andamento. Porém, resultados preliminares observados com sua utilização têm se mostrado promissores [Inacio and Dantas 2018b, Inacio and Dantas 2018a]. Espera-se, com a evolução da pesquisa, que o IORE possa auxiliar trabalhos na área de E/S e armazenamento paralelo de larga escala, não apenas no domínio de aplicações de CAD, mas também em aplicações de Big Data e Internet of Things. O código fonte da ferramenta está disponível gratuitamente no repositório do Laboratório de Pesquisa em Sistemas Distribuídos (LAPESD) no Github (`http://github.com/lapesd/iore`).

1.5. Considerações Finais

A proposta deste minicurso foi apresentar uma introdução a sistemas de E/S e armazenamento paralelos voltadas para ambientes de computação de alto desempenho (CAD). Mostrou-se que, cada vez mais, é preciso se preocupar com a forma como grandes conjuntos de dados são recuperados e armazenados por aplicações distribuídas de larga escala, visto que o acesso a estes dados pode ocupar um tempo significativo da execução da aplicação. Um visão geral da infraestrutura física e da pilha de software de E/S paralela tipicamente encontrada em ambientes de CAD modernos foi apresentada, identificando as principais funções de cada elemento e camada.

Em seguida, as principais características de sistemas de arquivos paralelos (SAPs) foram apresentadas. Estes sistemas, responsáveis por prover o sistema de armazenamento secundário em ambientes de grande porte, como clusters de larga escala e supercomputadores, são projetados para oferecer alta escalabilidade, em termos tanto de capacidade quanto de desempenho, e suportar acessos altamente concorrentes. A E/S paralela, da perspectiva das aplicações, também foi explorada neste minicurso. O fluxo básico para acesso a dados em um arquivo compartilhado por múltiplos processos de uma aplicação distribuída foi discutido considerando-se três APIs amplamente utilizadas: chamadas de sistema POSIX, funções do padrão C e funções independentes e coletivas do MPI-IO. Por fim, algumas das mais populares ferramentas para geração de carga de trabalho e avaliação de desempenho de E/S foram apresentadas. Estas ferramentas visam, em geral, auxiliar pesquisadores, usuários, desenvolvedores e administradores de sistemas a otimizar o desempenho da E/S em suas aplicações e ambientes.

Referências

[ANL 2002] ANL (2002). Parallel I/O Benchmarking Consortium. `http://www.mcs.anl.gov/research/projects/pio-benchmark/`.

[Baker et al. 2014] Baker, A. H., Xu, H., Dennis, J. M., Levy, M. N., Nychka, D., and Mickelson, S. A. (2014). A methodology for evaluating the impact of data compression on climate simulation data. In *HPDC '14 Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 203–214. ACM Press.

[Bell et al. 2009] Bell, G., Hey, T., and Szalay, A. (2009). Beyond the Data Deluge. *Science*, 323(5919):1297–1298.

[Boito et al. 2018] Boito, F. Z., Inacio, E. C., Bez, J. L., Navaux, P. O. A., Dantas, M. A. R., and Denneulin, Y. (2018). A Checkpoint of Research on Parallel I/O for High-Performance Computing. *ACM Computing Surveys*, 51(2):1–35.

[Braam and Schwan 2002] Braam, P. J. and Schwan, P. (2002). Lustre: The intergalactic file system. In *OLS '02 Proceedings of the Ottawa Linux Symposium*, pages 50–54.

[Carns et al. 2009] Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., and Riley, K. (2009). 24/7 Characterization of petascale I/O workloads. In *CLUSTER '09 Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE.

[CERN 2016] CERN (2016). Processing: What to record? `http://home.cern/about/computing/processing-what-record`.

[Chen et al. 2009] Chen, J. H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W. K., Ma, K. L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., and Yoo, C. S. (2009). Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):1–31.

[Chen et al. 1994] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. (1994). RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185.

[Dorier et al. 2016] Dorier, M., Sisneros, R., Gomez, L. B., Peterka, T., Orf, L., Rahmani, L., Antoniu, G., and Bougé, L. (2016). Adaptive Performance-Constrained In Situ Visualization of Atmospheric Simulations. In *CLUSTER '16 Proceedings of the IEEE International Conference on Cluster Computing*, pages 269–278. IEEE.

[Free Software Foundation 2018] Free Software Foundation (2018). The GNU C Library Manual. Technical report.

[Guzman et al. 2016] Guzman, J. C., Chapman, J., Marquarding, M., and Whiting, M. (2016). Status report of the end-to-end ASKAP software system: towards early science operations. In Chiozzi, G. and Guzman, J. C., editors, *Software and Cyberinfrastructure for Astronomy III*, volume 9913 of *SPIE Proceedings*. International Society for Optics and Photonics.

[Herbein et al. 2016] Herbein, S., Ahn, D. H., Lipari, D., Scogland, T. R., Stearman, M., Grondona, M., Garlick, J., Springmeyer, B., and Taufer, M. (2016). Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In *HPDC '16*

Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, pages 69–80. ACM Press.

[IBM 2018] IBM (2018). Overview of IBM Spectrum Scale. `https://www.ibm.com/support/knowledgecenter/en/STXKQY{\_}4.2.3/com.ibm.spectrum.scale.v4r23.doc/bl1ins{\_}intro.htm`.

[IEEE and The Open Group 2013] IEEE and The Open Group (2013). IEEE Std 1003.1-2013 - Standard for Information Technology - Portable Operating System Interface (POSIX). Technical report, IEEE.

[Inacio et al. 2017a] Inacio, E. C., Barbetta, P. A., and Dantas, M. A. R. (2017a). A Statistical Analysis of the Performance Variability of Read/Write Operations on Parallel File Systems. *Procedia Computer Science - Special Issue: International Conference on Computational Science, ICCS 2017*, 108:2393–2397.

[Inacio and Dantas 2014] Inacio, E. C. and Dantas, M. A. R. (2014). A Survey into Performance and Energy Efficiency in HPC, Cloud and Big Data Environments. *International Journal of Networking and Virtual Organisations*, 14(4):299–318.

[Inacio and Dantas 2018a] Inacio, E. C. and Dantas, M. A. R. (2018a). An I/O Performance Evaluation Tool for Distributed Data-Intensive Scientific Applications. In *LADaS '18 - Proceedings of the Latin America Data Science Workshop*, pages 9–16. CEUR-WS.

[Inacio and Dantas 2018b] Inacio, E. C. and Dantas, M. A. R. (2018b). IORE : A Flexible and Distributed I/O Performance Evaluation Tool for Hyperscale Storage Systems. In *ISCC '18 Proceedings of the IEEE Symposium on Computers and Communication*, page (to appear). IEEE.

[Inacio et al. 2015a] Inacio, E. C., Dantas, M. A. R., and de Macedo, D. D. J. (2015a). Towards a performance characterization of a parallel file system over virtualized environments. In *ISCC '15 Proceedings of the 20th IEEE Symposium on Computers and Communications*, pages 595–600. IEEE.

[Inacio et al. 2017b] Inacio, E. C., Nonaka, J., Ono, K., and Dantas, M. A. R. (2017b). Analyzing the I/O Performance of Post-Hoc Visualization of Huge Simulation Datasets on the K Computer. In *WSCAD '17 - Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 148–159. SBC.

[Inacio et al. 2015b] Inacio, E. C., Pilla, L. L. L., and Dantas, M. A. R. (2015b). Understanding the Effect of Multiple Factors on a Parallel File System's Performance. In *WETICE '15 Proceedings of the 24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 90–92. IEEE.

[Li et al. 2003] Li, J., Liao, W.-k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M. (2003). Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03 Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, New York, New York, USA. ACM Press.

[Liu et al. 2012] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., and Maltzahn, C. (2012). On the role of burst buffers in leadership-class storage systems. In *MSST '12 Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, pages 1–11. IEEE.

[LLNL 2013] LLNL (2013). IOR: Parallel filesystem I/O benchmark. `https://github.com/llnl/ior`.

[Lofstead et al. 2010] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M. (2010). Managing Variability in the IO Performance of Petascale Storage Systems. In *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE.

[Lüttgau et al. 2018] Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J., and Ludwig, T. (2018). Survey of Storage Systems for High-Performance Computing. *Supercomputing Frontiers and Innovations*, 5(1):31–58.

[Mitchell et al. 2011] Mitchell, C., Ahrens, J., and Wang, J. (2011). VisIO: Enabling Interactive Visualization of Ultra-Scale, Time Series Data via High-Bandwidth Distributed I/O Systems. In *IPDPS '11 Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 68–79. IEEE.

[Miyamoto et al. 2013] Miyamoto, Y., Kajikawa, Y., Yoshida, R., Yamaura, T., Yashiro, H., and Tomita, H. (2013). Deep moist atmospheric convection in a subkilometer global simulation. *Geophysical Research Letters*, 40(18):4922–4926.

[Moore et al. 2011] Moore, M., Bonnie, D., Ligon, W., Mills, N., Yang, S., Ligon, B., Marshall, M., Quarles, E., Sampson, S., and Wilson, B. (2011). OrangeFS: Advancing PVFS. In *FAST '11 Proceedings of the 9th USENIX conference on File and Storage Technologies*, pages 1–2, San Jose, CA, USA. USENIX Association.

[MPI Forum 1997] MPI Forum (1997). MPI-2: Extensions to the Message-Passing Interface. Technical report, Message Passing Interface Forum.

[MPIForum 2018] MPIForum (2018). Message Passing Interface Forum. `http://www.mpi-forum.org/`.

[Nagle et al. 2004] Nagle, D., Serenyi, D., and Matthews, A. (2004). The Panasas ActiveScale Storage Cluster - Delivering Scalable High Bandwidth Storage. In *SC '04 Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE.

[Nonaka et al. 2018] Nonaka, J., Inacio, E. C., Ono, K., Dantas, M. A. R., Kawashima, Y., Kawanabe, T., and Shoji, F. (2018). Data I/O management approach for the post-hoc visualization of big simulation data results. *International Journal of Modeling, Simulation, and Scientific Computing*.

[Nonaka et al. 2014] Nonaka, J., Ono, K., and Fujita, M. (2014). Multi-step image compositing for massively parallel rendering. In *HPCS '14 Proceedings of the International Conference on High Performance Computing & Simulation*, pages 627–634. IEEE.

[Omnibond 2018] Omnibond (2018). The OrangeFS Project. http://www.orangefs.org.

[OpenSFS 2018] OpenSFS (2018). Lustre. http://lustre.org/.

[Prabhat and Koziol 2014] Prabhat and Koziol, Q. (2014). *High Performance Parallel I/O*. Chapman & Hall/CRC, 1 edition.

[Radha et al. 2015] Radha, K., Rao, B. T., Babu, S. M., Rao, K. T., Reddy, V. K., and Saikiran, P. (2015). Service Level Agreements in Cloud Computing and Big Data. *International Journal of Electrical and Computer Engineering*, 5(1):158–165.

[Reed and Dongarra 2015] Reed, D. A. and Dongarra, J. (2015). Exascale computing and big data. *Communications of the ACM*, 58(7):56–68.

[Roten et al. 2016] Roten, D., Cui, Y., Olsen, K. B., Day, S. M., Withers, K., Savran, W. H., Wang, P., and Mu, D. (2016). High-Frequency Nonlinear Earthquake Simulations on Petascale Heterogeneous Supercomputers. In *SC '16 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.

[Shan et al. 2008] Shan, H., Antypas, K., and Shalf, J. (2008). Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE.

[Song et al. 2012] Song, H., Jin, H., He, J., Sun, X.-H., and Thakur, R. (2012). A Server-Level Adaptive Data Layout Strategy for Parallel File Systems. In *IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2095–2103. IEEE.

[Thakur et al. 1999] Thakur, R., Gropp, W., and Lusk, E. (1999). On implementing MPI-IO portably and with high performance. In *IOPADS '99 Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, New York, USA. ACM Press.

[Thakur et al. 2002] Thakur, R., Gropp, W., and Lusk, E. (2002). Optimizing noncontiguous accesses in MPI–IO. *Parallel Computing*, 28(1):83–105.

[The HDF Group 1997] The HDF Group (1997). Hierarchical Data Format, version 5. https://support.hdfgroup.org/HDF5/.

[Troppens et al. 2009] Troppens, U., Erkens, R., Mueller-Friedt, W., Wolafka, R., and Haustein, N. (2009). *Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS, iSCSI, InfiniBand and FCoE*. Wiley Publishing, 2 edition.

[Weil et al. 2006] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: a scalable, high-performance distributed file system. In *OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA. USENIX Association.

[Zhao et al. 2016] Zhao, D., Liu, N., Kimpe, D., Ross, R., Sun, X.-H., and Raicu, I. (2016). Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1824–1837.