Um Compressor de Arquivos Paralelo Compatível com o Bzip2

Vinicius Dalto do Nascimento[1]
Davi Vercillo [2]
Gabriel Pereira da Silva [3]
COPPE/Sistemas[1] – DCC/IM[2,3] – Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brasil
dalto@ufrj.br[1], davivercillo@dcc.ufrj.br[2], gabriel@dcc.ufrj.br[3]

Resumo

No cenário atual da computação verifica-se um aumento crescente da capacidade dos sistemas de armazenamento. Com isso, o desenvolvimento de ferramentas para a compressão rápida e eficiente de um grande número de arquivos, com tamanhos também cada vez maiores, se torna uma necessidade urgente. Simultaneamente, a ampla disponibilidade de recursos computacionais com múltiplos processadores, seja em um único computador, seja em um ambiente de rede, viabiliza o uso de aplicações paralelas para o atendimento dessa demanda. Este trabalho apresenta um compressor de arquivos paralelo, em que o trabalho de compressão é distribuído através de diversos processadores locais e remotos. São apresentadas duas versões desse compressor: uma que utiliza o paradigma de memória compartilhada e outra o de troca de mensagens. O uso de um servidor de arquivos paralelos, junto com rotinas do MPI-I/O, foi a solução encontrada para melhorar o desempenho do sistema de E/S, normalmente um gargalo nesse tipo de aplicação. Para verificar e validar o desempenho das implementações desenvolvidas, foram analisados diversos cenários e feitas comparações com os resultados de um compressor paralelo apresentado em um trabalho correlato.

1. Introdução

O uso de *softwares* para compressão de arquivos tornou-se uma ferramenta indispensável atualmente, tanto para um usuário normal que deseja comprimir fotos para enviar por correio eletrônico, quanto para um administrador de sistemas que deseja fazer uma cópia de segurança dos arquivos do seu servidor.

As ferramentas para compressão de arquivos já existem desde a década de 70, por isso encontram-se atualmente diversos programas com esse propósito. Dentre esses podemos citar o *gzip*[6], *bzip2*[11], *rar*[15], *ace*[14], que se destacam de outros menos conhecidos.

Os diversos algoritmos de compressão podem ser classificados em dois métodos: com perdas e sem perdas. No método sem perdas a compressão ocorre de forma que os arquivos de saída fiquem idênticos aos arquivos de entrada quando são descomprimidos. No segundo método, ao realizar a compressão, os arquivos sofrem perdas durante o processo. Para esse segundo caso, o arquivo descomprimido não ficará idêntico ao arquivo original de entrada.

Para certas aplicações, como fotos e vídeos, pequenas perdas não são tão importantes quanto obter-se uma melhor taxa de compressão. A grande vantagem da compressão com perdas é que esta proporciona grandes taxas de compressão, por essa razão ela é muito popular. No caso de arquivos como textos, programas executáveis e imagens médicas, a reconstrução exata do arquivo de entrada é essencial.

Um assunto que vem se tornando muito importante atualmente é como paralelizar essas ferramentas de compressão. Neste trabalho, dois protótipos de compressores paralelos de arquivos sem perdas são apresentados. O primeiro utiliza o paradigma da memória compartilhada e o segundo utiliza o paradigma da troca de mensagens combinado com o paradigma de memória compartilhada.

A implementação que serve de base para a paralelização faz uso do algoritmo **BWT**, embutido na biblioteca de compressão *libbzip2*[11]. A grande vantagem no uso da *libbzip2* é que o arquivo comprimido gerado pelo algoritmo paralelo fica compatível com a versão seqüencial da biblioteca, ou seja, não é gerado um arquivo proprietário na versão paralela do compressor.

O algoritmo de compressão utilizado na *libbzip2,* sobre o qual há diversos trabalhos publicados, é o **BWT** (*Burrows-Wheeler Transform*)[8]. O **BWT** divide o arquivo a ser comprimido em blocos com o mesmo tamanho. Cada um desses blocos sofre uma operação de codificação, de forma que caracteres (*bytes*) idênticos fiquem próximos uns dos

outros. Sobre esse bloco modificado é aplicado o algoritmo de *Huffman*, o que resulta em uma taxa de compressão melhor do que se apenas fosse aplicado o algoritmo de *Huffman* convencional. Em compensação o **BWT** possui um tempo de compressão maior. A grande vantagem do algoritmo **BWT** é que ele pode ser paralelizado, pois a leitura em blocos faz com que a compressão de cada bloco seja independente e possa ser feita simultaneamente, diminuindo o tempo total de compressão.

Atualmente, com a difusão dos processadores com mais de um núcleo, a paralelização de diversos algoritmos tem sido uma forma utilizada para aumentar o desempenho de várias aplicações. Assim, a paralelização do algoritmo **BWT** aparece como uma boa opção para acelerar a compressão de arquivos.

O paradigma de memória compartilhada é adequado para a paralelização em um único computador, onde todos os processadores têm acesso a uma memória global compartilhada. Contudo, isso limita o ganho máximo que pode ser obtido ao número de processadores daquele computador, que é normalmente de apenas algumas unidades.

Para aumentar o grau de paralelismo disponível é necessário que a implementação utilize o paradigma de troca de mensagens. Nesse caso, processadores residentes em computadores distintos se comunicam através de uma rede de comunicação e realizam a compressão em paralelo. Como o número de processadores envolvidos pode chegar a algumas centenas ou milhares, isso aumenta o potencial de paralelismo consideravelmente.

Aplicações de compressão fazem uso intensivo de acesso a disco. Quando existem várias aplicações executando em diversos computadores, fazendo o uso da rede de comunicação para acesso ao sistema de arquivos, ocorre um processo de contenção, pois vários computadores fazem acesso simultâneo a um mesmo arquivo em um único computador. Isso logo se torna um gargalo e torna seqüencial o acessos aos arquivos, diluindo todos os ganhos obtidos com a paralelização da compressão.

Para diminuir os efeitos causados por essa contenção, foi utilizado o sistema de arquivos paralelo *PVFS2 (Parallel Virtual File System 2)*[1] nos experimentos realizados. O sistema de arquivos *PVFS2* divide os arquivos em diversas partes que são distribuídas por vários servidores, de modo que os acessos aos arquivos são feitos de uma forma não centralizada. Com o acesso distribuído, os gargalos causados pelos acessos simultâneos ao disco são atenuados, permitindo um melhor desempenho do compressor paralelo.

Este artigo está organizado da seguinte maneira: na seção seguinte apresentam-se alguns trabalhos relacionados com o tema. Na Seção 3 é descrevida a implementação do compressor apresentado neste artigo. Já na Seção 4 é feito um detalhamento dos experimentos realizados. Na Seção 5 são apresentadas as conclusões e trabalhos futuros.

2. Trabalhos Correlatos

Outros trabalhos também abordaram a compressão de arquivos de forma paralela, como [3] e [4].

No trabalho de Jeff Gilchrist [3] existem também duas implementações, uma utilizando memória compartilhada e outra troca de mensagens. A implementação utilizando memória compartilhada foi codificada utilizando a biblioteca de criação de emphthreads chamada *pthreads* [9], onde para cada *thread* é passado um ponteiro para um método que será executado pela *thread*. Todo o controle de acesso aos recursos compartilhados pelas *threads* é de responsabilidade do programador, tendo este que fazer uso de primitivas de sincronização para acesso às variáveis compartilhadas.

A diferença na implementação apresentada neste artigo é que a programação paralela é feita através de diretivas de compilação fornecidas pelo *OpenMP* que estão disponíveis no compilador *gcc* a partir da versão 4.1.

Além da implementação utilizando a biblioteca *libbzip2*, o trabalho de Jeff Gilchrist também utilizou uma implementação feita em C++ do algoritmo do BWT chamada de *bwtzip* [5].

3. Implementações do Compressor

Serão apresentados e analisados neste artigo duas implementações do compressor de arquivos paralelo. Na primeira delas foi utilizado o paradigma de memória compartilhada e na outra o paradigma de troca de mensagens associado ao de memória compartilhada.

Na implementação utilizando memória compartilhada foi utilizado o padrão de programação *OpenMP*[10]. O *OpenMP* é uma API que permite a programação paralela em C/C++ e Fortran utilizando o paradigma de programação de memória compartilhada com o suporte em várias plataformas, com processsadores e sistemas operacionais de diversos tipos. Essa API foi definida em conjunto por um grupo de fabricantes de *hardware* e *software*, o *OpenMP* é portável e escalável, podendo ser utilizado tanto em computadores *desktops* como em supercomputadores.

Existem outras bibliotecas conhecidas para criação de *threads* paralelas, como por exemplo *pthreads*[9], mas a opção pelo *OpenMP* foi baseada na simplicidade e facilidade de programação.

Na implementação utilizando o paradigma de troca de mensagens, foi utilizado o *OpenMPI*. O *OpenMPI* é uma implementação aberta do padrão **MPI-2** ***(M**essage **P**assing **I**nterface)*[7] que é desenvolvido e mantido por instituições acadêmicas em parceria com a indústria. É um padrão suportado em linguagens de programação Fortran e C/C++; sistemas operacionais de 64 e 32 bits; redes heterogêneas; além de permitir entrada e saída remota.

A função do **MPI** é construir uma topologia virtual e permitir a sincronização e comunicação entre um conjunto de processadores (podem ser chamados também de nodos ou máquinas na rede). O **MPI**, ao contrário do *OpenMP*, sempre trabalha com processos. Geralmente é disparada uma cópia de processo para cada processador, onde estes processos podem se comunicar através de troca de mensagens.

Existem primitivas no **MPI** que conseguem "dividir" o trabalho entre os processadores encontrados na rede. Essa divisão ocorre chamando métodos como, por exemplo, o `MPI_Scatter`[7] que divide um conjunto de blocos memória em seqüência, em pedaços que são enviados para outros processadores através da rede.

3.1. Paradigma de Memória Compartilhada

A paralelização utilizando memória compartilhada tem como foco obter o maior rendimento em máquinas com mais de um processador ou com processadores com vários núcleos. A idéia é disparar uma *thread* "compressora", que de fato irá fazer o trabalho pesado, em cada núcleo. No início do programa são feitas atribuições iniciais e alocação de memória para uso geral da aplicação, seguidas de uma seção com as diretivas de paralelização do *OpenMP*. Nessa região paralela, são criadas no mínimo 3 *threads*. Elas podem ser do tipo:

- Thread Leitora - faz a leitura de blocos de dados com tamanho fixo do arquivo a ser comprimido;
- Thread Compressora - faz a compressão de cada um dos blocos;
- Thread Escritora - escreve os blocos comprimidos no arquivo de saída.

Em cada tarefa de compressão é necessário que haja no mínimo, uma *thread leitora*, uma *thread compressora* e uma *thread escritora*. O número de *threads compressoras* pode ser especificado pelo usuário, pois são elas que efetivamente farão o trabalho de compressão, sendo que o ideal é que exista uma *thread compressora* por núcleo de processador disponível. Já as *threads leitoras* e *escritoras,* que não demandam uso intensivo de processador, são executadas com apenas uma cópia apenas de cada.

Como pode existir mais de uma *thread compressora*, é necessário que seja feita uma exclusão mútua para o acesso concorrente das *threads* ao *buffer* de leitura. Ou seja, o incremento do ponteiro do *buffer* é feito com uma operação atômica indivisível. Para realizar essa operação atômica, foi utilizado a diretiva `#pragma omp atomic`[10] do *OpenMP*.

Um detalhe importante, é que somente a *thread compressora* pode alocar o espaço para o bloco comprimido, pois o resultado da compressão de um bloco pode ser menor ou maior que o seu tamanho original. O bloco poderá ficar maior que o bloco sem compressão caso não haja padrões de repetição de caracteres (*bytes*) no bloco.

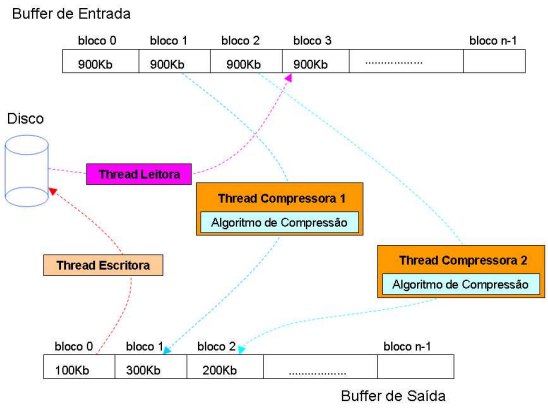

Figura 1. Modelo de Paralelização com Memória Compartilhada

Ao escrever o bloco comprimido no *buffer* de saída não é necessário realizar nenhuma operação especial de ordenação, porque a estrutura de dados na qual o bloco comprimido está contido, contém a ordem com que o bloco foi lido do arquivo de entrada. Assim a *thread compressora* sabe exatamente a posição do *buffer* que o bloco deverá ser escrito. Um exemplo com 2 *threads compressoras* é mostrado na Figura 1.

A *thread escritora* verifica sempre se a próxima posição do *buffer* é diferente de `NULL`, pois isso indica que alguma *thread compressora* escreveu naquela posição. Caso a posição do *buffer* ainda seja igual a `NULL`, a *thread escritora* executa uma função `sleep` com *delay* de 1 ms para deixar de ocupar o processador. Quando o tempo do `sleep` termina, é feita novamente a verificação, e essa operação se repete até que todos os blocos tenham sido escritos no disco. Repare que poderia ter sido utilizado um semáforo para sincronização das *threads*, mas a princípio o uso de *sleeps* pareceu mais simples para uma primeira implementação, mas o que futuramente poderá ser um caso de estudo de comparação.

3.2. Troca de Mensagens

Na implementação utilizando troca de mensagens, o código base é o mesmo da implementação para memória compartilhada, adicionando-se as primitivas de comunicação do **MPI**. Nesse caso, uma cópia de cada processo é executada em cada uma das máquinas da rede, cada processo com um conjunto de *threads* para realizar o trabalho de compressão, leitura e escrita. Se houver mais processadores disponíveis em cada máquina, eles poderão ser uti-

lizados em todo o seu potencial.

Um grande problema ao se trabalhar com troca de mensagens é gerenciar os acessos de escrita ou de leitura de diversos processos a um mesmo arquivo. Para resolver este problema, foi utilizada a versão 2 do *MPI* que dá suporte a escrita e leitura compartilhada por múltiplos processos.

Para ler os arquivos a serem comprimidos, é necessário fazer um acesso pela rede até a máquina onde esses arquivos estão armazenados. O mesmo ocorre quando é necessário escrever o resultado da compressão. Acessos através da rede a arquivos remotos normalmente são operações demoradas, que levam mais tempo do que acessos ao disco local.

Adicionalmente, diversos processos farão requisições simultâneas à máquina que possui o arquivo (de leitura ou escrita), resultando em contenção na máquina hospedeira do arquivo. Este caso acontece com freqüência quando utiliza-se um sistema de arquivos distribuído como *NFS (**N**etwork **F**ile **S**ystem)*[12], por exemplo. Para melhorar a eficiência dos acessos aos arquivos para leitura e escrita, escolhemos o **PVFS2** *(**P**arallel **V**irtual **F**ile **S**ystem **2**)* nos experimentos realizados.

O sistema de arquivos paralelo **PVF2** distribui um arquivo por vários discos em vários servidores. Isso diminui bastante o problema da contenção que surge com o acesso e aumenta também o *throughput* no acesso a um mesmo arquivo, seja este acesso feito por um único ou vários clientes.

Para realizar uma leitura de um bloco no arquivo, basta invocar o método `MPI_File_read_at`[7] do **MPI**. Este método lê um número de *bytes* especificado na passagem por parâmetro na posição do arquivo especificada por um *offset*.

Com o uso deste método de acesso ao arquivo, os processos acessam o arquivo de modo independente, ou seja, não necessitam que um processo aguarde outro processo terminar de ler para poder iniciar a sua leitura. No caso da escrita, o mesmo método não pode ser adotado, pois, para que o arquivo de saída seja compatível com a biblioteca *libbzip2*, a ordem de escrita dos blocos no disco precisa ser a mesma que no arquivo original.

O método do **MPI** utilizado para garantir a ordem de escrita dos blocos é o `MPI_File_write_ordered`[7]. Este método faz com que os processos escrevam os blocos em ordem, por exemplo, suponha que existem 7 blocos de tamanhos variáveis e 3 processos para serem escritos no arquivo. Utilizando o método `MPI_File_write_ordered` a ordem de escrita dos processos sempre será {P0, P1, P2, P0, P1, P2, ... } até que todo o arquivo tenha sido escrito no disco. Um detalhe importante, é que se o processo 2 já tem o bloco pronto para escrever no disco, mas o processo 1 ainda não escreveu no disco, o processo 2 precisa aguardar que o processo 1 escreva o seu bloco no disco. Essa espera ocasiona uma pequena perda de desempenho, que é compensada pela compatibilidade final do arquivo com aquele gerado por programas que usam a biblioteca *libbzip2*.

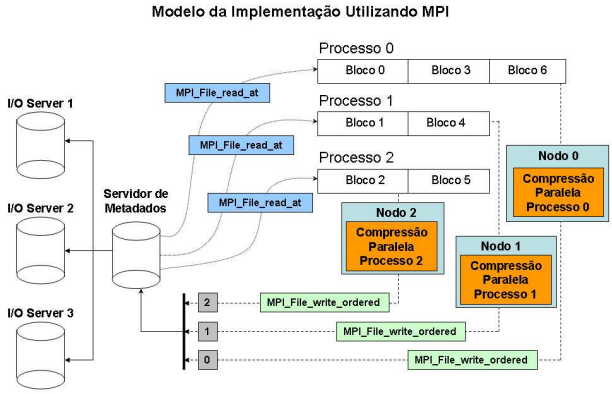

Figura 2. Modelo com uso de MPI

Como o algoritmo do **BWT** nos permite comprimir os blocos independentemente, a implementação utilizando **MPI** foi facilitada pelo uso do **MPI I/O**. A modificação feita nessa versão do compressor foi adicionar as chamadas iniciais do **MPI**, chamar a leitura por *offsets* e chamar a escrita do **MPI**. Assim, são disparadas cópias da versão com *threads* em cada nodo da rede, obtendo assim melhor aproveitamento de todos os recursos disponíveis. A Figura 2 ilustra com mais detalhes o funcionamento da aplicação utilizando **MPI**.

Na Figura 2, os blocos são lidos dos servidores do *PVFS2* pelos processos, através da rotina `MPI_File_read_at`. Em seguida, é aplicado o algoritmo de compressão sobre os blocos. Por último, os blocos são escritos nos servidores do *PVFS 2* através da rotina `MPI_File_write_ordered` que é chamada pelos processos.

Repare que na linha pontilhada, onde é chamada a rotina `MPI_File_write_ordered`, incidirá em uma barreira com os números 0, 1 e 2. Esses números representam a ordem em que os processos revezarão para escrever os seus blocos nos servidores. Por exemplo, o primeiro a escrever será o processo 0, o segundo será o processo 1 e o terceiro será o processo 2, em seguida será o processo 0 novamente, mantendo assim esta ordem até que todos os blocos tenham sidos escritos nos discos dos servidores.

4. Experimentos

Foram realizados dois tipos de experimentos em ambientes de execução distintos. O primeiro experimento consistia em comprimir um arquivo em uma máquina local, onde esta máquina possuía dois processadores com 4 núcleos cada. Os arquivos de leitura e escrita estão no disco

local da máquina. O parâmetro variado neste experimento foi o número de *threads* executadas simultaneamente.

O segundo experimento foi feito utilizando-se um total de 16 máquinas, 12 para execução da aplicação e 4 para fazer o papel de servidore E/S do sistema de arquivos distribuído **PVFS2**. Neste experimento o foco era variar o número de máquinas para avaliar o *speedup*, mantendo-se fixo o número de *threads* por nodo.

O tipo de arquivo utilizado nos testes foi um arquivo texto qualquer, que era concatenado sucessivamente com ele mesmo para obter arquivos maiores. Desta forma, quando o tamanho do arquivo era dobrado, o seu padrão de repetições aumentava e, conseqüentemente, fazia com que sua taxa de compressão aumentasse. Este método nos fez poupar espaço em disco e nos permitiu fazer uma comparação bit a bit para verificar a integridade do arquivo comprimido. Esta operação de comparação é demorada em arquivos muito grandes, mas como a taxa de compressão foi alta, os arquivos comprimidos se tornavam pequenos, permitindo assim a comparação.

4.1. Experimentos com Memória Compartilhada

Os experimentos utilizando a versão do compressor que utiliza somente memória compartilhada foram feitos de duas formas. A primeira comprime arquivos de tamanhos diferentes variando o número de *threads compressoras* em uma máquina com dois processadores de 4 núcleos. A segunda faz a comparação com o compressor apresentado no trabalho de [3], feito com *pthreads*.

No primeiro experimento o número de *threads* compressoras é variado para encontrar uma configuração com melhor desempenho para a máquina utilizada nos testes. Nesse experimento não há como controlar em qual processador cada *thread* irá ser executada. Na aplicação existem no mínimo 3 *threads*, uma *leitora*, uma *escritora* e uma *compressora*.

Quando a compressão é disparada apenas com as 3 *threads* necessárias, não há como controlar em qual processador cada *thread* irá executar. Essa escolha é uma tarefa do sistema operacional que depende da sua política de escalonamento. Por exemplo, se houver 3 *threads* e um processador com 2 núcleos, fica a critério do sistema operacional escolher qual *thread* executará em cada núcleo, sendo que a *thread* sobressalente geralmente executará no núcleo que tiver menos trabalho, quando houver a troca de contexto.

O *hardware* disponível para realizar os experimentos era o seguinte:

- 2 Processadores Intel Xeon com 4 núcleos de 2.66 GHz e Memória cache de 12 Mbytes
- 16 Gbytes de memória RAM
- Disco de 160 Gbytes

Todos os experimentos com memória compartilhada foram executados nesta máquina. Como o processador tinha dois processadores com 4 núcleos, nos seria conveniente testar a aplicação com até 8 *threads* compressoras, pois com 8 núcleos, em teoria, cada *thread* compressora realizaria seu trabalho pesado em um núcleo diferente.

Os arquivos utilizados para efetuar os testes de compressão foram arquivos de texto gerados a partir da concatenação de um arquivo original, como citado na seção anterior. Os tamanhos de arquivos eram de 128 Mbytes, 256 Mbytes, 512 Mbytes, 1 Gbytes, 2 Gbytes e 4 Gbytes. Os arquivos se encontravam no disco local, de onde eram lidos pela aplicação, e a aplicação escrevia no mesmo disco.

Na Tabela 1 pode-se ver os tempos de execução do compressor com memória compartilhada implementado com *OpenMP*. Na coluna 1, onde temos o label "TCs", representa o número de *threads compressoras* utilizado na execução. Na linha 2, se encontram os respectivos tamanhos de arquivos utilizados na execução, seguindo na coluna abaixo destes, os respectivos tempos de execução. Por exemplo, no experimento ao comprimir um arquivo de 1 Gbytes e utilizando 4 *threads compressoras*, obteve-se um tempo de execução igual a 154 segundos.

Tabela 1. Tempos de Execução em Segundos

	Tamanho dos Arquivos (MBytes)					
TCs	128	256	512	1024	2048	4096
1	61	123	309	617	1261	2528
2	31	63	155	311	621	1241
4	16	31	78	154	309	617
6	13	26	65	130	259	519
8	12	22	58	118	233	461
10	11	22	55	108	215	434

Ao analisar os resultados da Tabela 1, pode-se verificar, por exemplo, que o resultado do tempo de compressão de um arquivo de 4 Gbytes é igual a 2528 segundos para ser comprimido por uma única *thread*. Mas, quando utilizam-se 8 *threads* por exemplo (uma para núcleo), a compressão foi efetuada em 461 segundos, o que é equivalente a apenas 18% (aproximadamente) do tempo de execução usando uma *thread*.

Observa-se uma diminuição nos tempos de compressão utilizando até 10 *threads compressoras*. Isso significa que ainda há duas *threads* adicionais, uma para leitura outra para escrita dos blocos, e ainda há o ganho no tempo de execução, mesmo com apenas 8 núcleos de processamento disponíveis. O que parece indicar que as *threads* de E/S realmente fazem pouco uso de processamento. Nota-se que mesmo as *threads* compressoras não ocupam os núcleos 100% do tempo, permitindo assim que ainda haja diminuição nos tempos de execução, mesmo quando o número de *threads* compressoras é maior que o número de

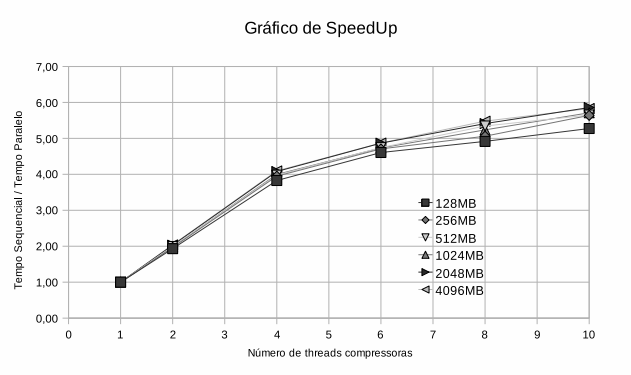

Figura 3. Speedup com o OpenMP

núcleos disponíveis para processamento.

Realizamos testes para um número maior de *threads* e com o número de *threads* simultâneas maior que 10 não há redução significativa no tempo de processamento, que, ao contrário, passa a aumentar com o uso de um número excessivo de *threads* compressoras devido provavelmente à sobrecarga de troca de contexto e disputa por recursos do sistema.

O gráfico da Figura 3 mostra o comportamento do *speedup* da execução. O comportamento está dentro do esperado, sendo que o *speedup* máximo obtido ficou em torno de 5,8. Isso corresponde a uma eficiência de cerca de 73% para 8 núcleos de processamento.

O *speedup* é linear até cerca de 4 *threads* compressoras executando em paralelo, quando então começa haver uma diminuição na eficiência. Supomos que isso se deva à arquitetura do processador de 4 núcleos, que começa a apresentar alguma contenção no barramento (comum para todos os núcleos) quando do acesso à memória principal e, eventualmente, diminuição nas taxas de acerto na cache de nível 2, que é também compartilhada por todos os núcleos.

O *speedup* é maior quanto maiores forem os arquivos a serem comprimidos, o que mostra que a aplicação escala bem, sendo os tempos gastos com compressão mais significativos que os tempos gastos com a realização de E/S.

Com os experimentos apresentados chegamos a algumas conclusões sobre o compressor analisado. A primeira é que para se obter um bom desempenho, é necessário que o tempo de execução das *threads compressoras* sejam próximos, pois o atraso de uma delas fará com que o tempo de execução seja aumentado para o pior caso. A segunda conclusão é que o desempenho também dependerá número de *threads compressoras* que for ajustado para um processador com um determinado número de núcleos. Os melhores resultados foram obtidos quando destinou-se pelo menos uma *thread compressora* para núcleo de processador disponível.

Como pode-se ver na Tabela 1 mostrada anteriormente, o caso em que obteve-se o melhor desempenho foi com 10 *threads compressoras*, pois provavelmente, o sistema operacional pode manter os núcleos ocupados 100% do tempo e as *threads* de E/S não tem grande demanda por processamento.

Sabendo qual a melhor configuração para uma máquina com 2 processadores de 4 núcleos, mais um experimento para determinar a qualidade da implementação foi realizado. Para este propósito utilizou-se o compressor paralelo também baseado na *libbzip2* chamado *pbzip2*. O *pbzip2* pode ser encontrado no trabalho de [3]. Para realizar essa comparação os mesmos arquivos do experimento anterior foram comprimidos com o compressor do trabalho citado usando 8 *threads* compressoras. Na Tabela 2 podem-se ver os resultados:

Tabela 2. Comparando os resultados

	Tamanho dos Arquivos (Mbytes)					
Compressor	128	256	512	1024	2048	4096
pbzip2	12	23	57	111	228	453
ompbzip2	**12**	**22**	**58**	**118**	**233**	**461**

Pelo resultados apresentados na Tabela 2 verifica-se que os tempos de compressão estão muito próximos, com uma leve vantagem para o compressor da implementação de [3]. Esses resultados atestam a qualidade da implementação com uso de OpenMP, que possui um desempenho similar à implementação com *pthreads*, mas com a vantagem de possuir um número bem menor de linhas de código, sendo programada com muito mais facilidade.

4.2. Experimentos Utilizando Troca de Mensagens

Esse experimento consistiu em comprimir diversos arquivos de diferentes tamanhos, sendo que cada arquivo será desmembrado em conjuntos de blocos e serão comprimidos em máquinas diferentes. Foram variados o número de máquinas, de forma que fosse possível medir o tempo de execução para cada conjunto de máquinas. Foram utilizadas ao todo, 16 máquinas com a seguinte configuração:

- Processador Pentium IV de 3.0 GHz com tecnologia Hyperthread
- 1 GByte de memória RAM
- Rede Ethernet de 100 Mbps

As 16 máquinas não foram utilizadas para execução da aplicação, apenas 12 foram utilizadas para esse propósito. As outras 4 máquinas foram utilizadas como o servidor do *PVFS 2*, sendo que uma foi utilizada como *servidor de metadados* e as outras três foram utilizadas como *servidor de dados*.

Nas máquinas destinadas à execução da aplicação, foram disparados um processo por máquina via *MPI*. Cada processo foi configurado para trabalhar com duas *threads* com-

pressoras. Optou-se por essa configuração pois a tecnologia *hyperthread* permite otimizações nas execuções de *threads*, "simulando" assim um processador com dois núcleos.

Os arquivos utilizados para efetuar a compressão, eram arquivos textos gerados a partir da concatenação com eles mesmos, isso foi feito para aumentar o seu tamanho, neste caso os seu tamanho era dobrado. Os arquivos antes de serem comprimidos encontravam-se nos servidores do *PVFS2* com o propósito de aumentar a taxa de leitura e escrita. Os tamanhos de arquivos que foram utilizados foram de 128 Mbytes, 512 Mbytes, 1 Gbyte, 2 Gbytes e 4 Gbytes.

Nos experimentos foram utilizados 12 máquinas para efetuar a compressão dos arquivos. Foram medidos os tempos de compressão de cada arquivo citado anteriormente utilizando o conjunto de máquinas de tamanho variável. Os conjuntos tinham tamanho de 1, 2, 4, 6, 8, 10 e 12 máquinas.

Na Tabela 3 são apresentados os tempos de execução do compressor com uso de **MPI**. A coluna 1 representa o número de máquinas utilizadas para efetuar a compressão, enquanto que na linha 2 são mostradas os tamanhos de arquivos utilizados. Por exemplo, quando comprimiu-se um arquivo de 1 Gbyte utilizando 8 máquinas, gastaram-se 414 segundos para efetuar a compressão.

Tabela 3. Tempos de Execução em Segundos

	Tamanho dos Arquivos (Mbytes)				
Máquinas	128	512	1024	2048	4096
1	391	1387	2937	5432	11127
2	195	714	1482	3055	5882
4	100	377	774	1514	3062
6	75	270	519	1019	2205
8	62	216	414	824	1678
10	61	172	345	692	1271
12	**51**	**153**	**289**	**566**	**1085**

Pode-se verificar na Tabela 3 que ao aumentar o número de processadores, claramente o tempo de execução diminui. Para ilustrar, pode-se utilizar o exemplo mais visível, quando comprimiu-se um arquivo de 4 Gbytes em apenas uma máquina, foram gastos 11127 segundos, ou cerca de 185 minutos. Quando compara-se com a execução utilizando 12 máquinas, o tempo gasto foi 1085 segundos, o que equivale aproximadamente a 18 minutos, ou seja, o tempo de execução utilizando 12 máquinas foi menor que 10% do tempo de execução utilizando-se apenas uma máquina.

Verifica-se no gráfico da Figura 4 que o *speedup* tenta se aproximar de uma reta, diminuindo tempo de execução sempre que adicionamos mais máquinas para efetuar a compressão. Claro que o ideal seria que $E_p = \frac{S_p}{p} = 1$, mas dificilmente essa tendência será mantida quando aumentamos significativamente o tamanho do problema e o número

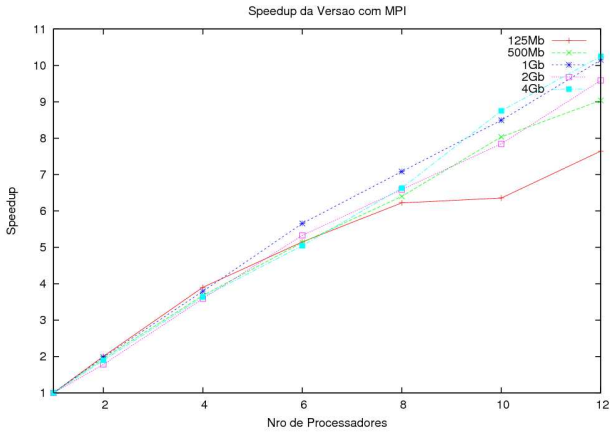

Figura 4. Speedup com Troca de Mensagens

de máquinas para resolve-lo. Com aumento de máquinas haverá um aumento no custo de comunicação, devido ao grande número de pacotes que irá circular pela rede. Mas mesmo assim, o ganho de tempo ao aumentar o número de máquinas é significativo.

5. Conclusões

A compressão de arquivos de grande tamanho é um problema cujo perfil aparentemente não seria adequado para a paralelização, por apresentar uso intensivo de E/S. Apesar disso, apresentamos nesse artigo resultados que demonstram que esse problema pode ser resolvido de maneira eficiente com uso de técnicas simples de programação paralela, associado ao uso de um sistema de arquivos paralelos.

Nos experimentos utilizando memória compartilhada percebe-se que a aplicação escala muito bem enquanto o número de *threads* é menor ou igual à metade dos processadores na máquina utilizada para os testes. A partir daí há uma diminuição da eficiência até um valor de 73% quando todos 8 processadores estão ocupados. Acreditamos que essa redução se deva a uma contenção no uso dos recursos compartilhados dos processadores tais como memória cache e barramento de acesso à memória.

Entretanto, esses são resultados são bastante animadores, pois há uma obtenção de ganho em uma aplicação com grande uso de E/S. Não esperava-se obter esse nível de eficiência quando foi iniciado esse projeto. Mesmo com o número de *thread* compressoras ligeiramente maior que o número de núcleos disponíveis para processamento, ainda assim houve diminuição nos tempos de compressão.

Outro aspecto positivo é que a implementação de compressor analisada mostrou-se robusta e escalável, comprimindo arquivos relativamente grandes, mantendo o seu desempenho e a compatibilidade com os arquivos gerados pela

aplicação sequencial *bzip2*.

A comparação com a implementação usando *pthreads* mostrou que os tempos de compressão de ambas a versões ficaram bem próximos, com a vantagem do código com uso de OpenMP ser mais simples e fácil de ser gerado.

Analisando os experimentos realizados na implementação utilizando troca de mensagens, verifica-se que ao adicionar mais máquinas aos experimentos, o tempo de compressão diminuía consideravelmente, tendendo a um *speedup* linear. Nos casos com muitas máquinas e tamanho de arquivo pequeno o *speedup* diminuía, pois o maior número de máquinas fazia com que os gastos com comunicação sejam mais significativos com relação ao tempo de execução. A versão com troca de mensagens do compressor também mostrou-se robusta e escalável nos experimentos realizados, apesar do reduzido número de máquinas.

Com os resultados de ambos os experimentos constatou-se que 8 máquinas de 3.0 GHz interligadas por uma rede não obtiveram um desempenho superior a uma máquina com um dois processadores de 4 núcleos de 2.6 GHz. Este fato ocorre porque a E/S em um disco local é bem mais rápida que a E/S através de um sistema de arquivos paralelos, já que a velocidade da rede utilizada nos testes era de 100 Mbps.

Como trabalho futuro pretendemos estender os experimentos para um sistema com um maior número de máquinas e também uma rede de interconexão mais rápida. Ainda, na versão com MPI, desejamos usar máquinas com maior número de processadores por cada nó e também um sistema de E/S paralelo com maior *throughput*.

Pretendemos estudar também algoritmos alternativos de compressão e variações, por exemplo, no tamanho dos blocos de compressão para obter maior eficiência tanto nas versões paralelas com OpenMP como também com MPI.

O código fonte do programa está disponível para aqueles que se mostrarem interessados no endereço apontado na referência [2].

Referências

[1] Parallel virtual file system - version 2, 2005. Disponível em: <http://www.pvfs.org/>. Acesso em Fevereiro, 2008.

[2] V. D. do Nascimento. Paralle file compressor, 2008. Disponível em: <http://sourceforge.net/projects/parallelfilecom/>. Acesso em Fevereiro, 2008.

[3] J. Gilchrist and A. Cuhadar. Parallel lossless data compression based on the burrows-wheeler transform. In *AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, pages 877–884, Washington, DC, USA, 2007. IEEE Computer Society.

[4] J. Kitzman and G. I. Fujiwara. Parallel file compression, 2005. Disponível em: <http://beowulf.lcs.mit.edu/18.337-2005/projects/compressionwriteup.pdf>. Acesso em Fevereiro, 2008.

[5] S. T. Lavavej. Bwtzip c++ implementation, 2006. Disponível em: <http://nuwen.net/bwtzip.html>. Acesso em Fevereiro, 2008.

[6] J. loup Gailly and M. Adler. Gzip, 2003. Disponível em: <http://www.gzip.org/>. Acesso em Fevereiro, 2008.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard 2.1*, 2008. Disponível em: <http://www-unix.mcs.anl.gov/mpi/www/>. Acesso em Junho, 2008.

[8] M. Nelson. Data compression with the burrows-wheeler transform. *Dr. Dobb's Journal*, 21(9):46–50, September 1996.

[9] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly, 1996.

[10] OpenMP Architecture Review Board. *OpenMP Application Program Interface 3.0*, 2008. Disponível em: <http://www.openmp.org/mp-documents/spec30.pdf>. Acesso em Maio, 2008.

[11] J. Seward. The bzip2 and libbzip2 official homepage, 2002. Disponível em: <http://www.bzip.org/>. Acesso em Fevereiro, 2008.

[12] Sun Microsystems, Inc. *NFS Version 3 Protocol Specification*, 2005. Disponível em: <http://www.faqs.org/rfcs/rfc1813.html>. Acesso em Fevereiro, 2008.

[13] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17. USENIX Association, 2008.

[14] winace. Ace, 2008. Disponível em: <http://www.winace.com/>. Acesso em Fevereiro, 2008.

[15] winrar. Rar, 2005. Disponível em: <http://www.rarlab.com/>. Acesso em Fevereiro, 2008.