Improving Middleware Performance with AdOC: an Adaptive Online Compression Library for Data Transfer

Emmanuel Jeannot
LORIA, Université H. Poincaré Nancy - 1
Vandœuvre les Nancy, France
Emmanuel.Jeannot@loria.fr

Abstract

In this article, we present the AdOC (Adaptive Online Compression) library. It is a user-level set of functions that enables data transmission with compression. The compression is performed dynamically during the transmission and the compression level is constantly adapted according to the environment. In order to ease the integration of AdOC into existing software the API is very close to the read and write UNIX system calls and respects their semantic. Moreover this library is thread-safe and is ported to many UNIX-like systems. We have tested AdOC under various conditions and with various data types. Results show that the library outperforms the POSIX read/write system calls on a broad range of networks (up to 100 Mbit LAN), whereas on Gbit Ethernet, it provides similar performance.

1. Introduction

Computational and data grids are distributed architectures that interconnect a set of heterogeneous computers (from a parallel machine to a desktop PC) with various types of networks (Internet, WAN, LAN, etc.). The objective of such grids is to gather distributed resources (CPU, disk, memory, etc.), to solve problems that require huge amount of computation or storage. Nowadays many middlewares [5, 6, 9, 17] are under development to allow applications to use grids in a transparent way. These middlewares manage the infrastructure, schedule the jobs, handle communications and data. In order to do this each middleware has to rely on a set of services (scheduling, accounting, resource discovery, etc.). In this paper we propose and describe a new service for grid middlewares and data transfer tools that enables compression on the fly for efficient transmission. The motivation for this work is that many (grid) applications require a large amount of data to be transmitted. In some cases, data transmission is the most time consuming part and therefore needs to be optimized.

The service we propose here is a library called AdOC (Adaptive Online Compression), which offers the possibility to transfer data while compressing it. It is an *adaptive* service as the compression level is dynamically changed according to the environment and the data. The adaptation process is required by the heterogeneous and dynamic nature of grids. For instance if the network is very fast, time to compress the data may not be available. But, if the visible bandwidth decreases (due to some congestion on the network), some time to compress the data may become available.

In this paper, AdOC is tested on several kind of networks and with different types of data. We compare AdOC read and write functions to the standard POSIX read and write system calls. We show that the latency of AdOC is similar to that of the POSIX read/write. We show that AdOC enables an increase of bandwidth depending on the data sent and the network (up to 6 times faster). We provide a conservative approach of compression that leads to no performance degradation on most kinds of networks (on Gbit Ethernet some microseconds are lost with AdOC) and any kind of data (even an incompressible one).

The API of the AdOC library is very close to the POSIX read/write system calls and respects their semantic. Therefore, it has been easily integrated into the NetSolve middleware [6]. The evaluation of the enhanced version of NetSolve shows a significant increase of performance on various scenarii whereas, on worst-case scenarii, no performance degradation is seen.

2. Adaptivity issues

Compression is often proposed in various transmission protocols such as FTP [15], PPP [16] or in the secure copy tool (`scp`). However, compressing is never the default behavior because no adaptation is provided. In some cases, it is worth to compress but not always.

In this paper, adapting means changing the compression level during the transmission. The *compression level* refers

on how efficiently data are compressed. Adapting the compression level (and in some cases disabling the compression) must be performed according to the following parameters:

- *Current speed of the network*. If the network is very fast, there is no time to compress the data. If the network is slow enough, some time may be available to compress the data. Moreover, the network is often shared by other users. Thus, its speed can change with the time: it is then required to change the compression level.
- *Current speed of the machine* on each side of the transmission. Compressing and uncompressing data requires some computational power. Before enabling compression, one must be sure that machines in both ends have enough computational power to perform the compression/decompression, without slowing down the transmission. Indeed, if it requires more time to compress, send and uncompress the data than just send the data uncompressed no gain can be expected. Moreover, many machines run multi-task operating systems (for instance UNIX). Therefore, the available CPU power may change with the time. In this case, it is required to adapt the compression level to the new conditions.
- *Size of the data to be transmitted*. Enabling compression adds a startup time (latency) to the transmission. Therefore, if small messages are to be sent, the startup time can be greater than the gain obtained with compression. Hence, for small data, the compression must be disabled.
- *Type of the data to be transmitted*. Some data are easier to compress than other. ASCII data compresses better and requires less time to compress than binary data. Moreover, for some files (such as directories archive), the nature of the data changes along the file. Hence, the compression level must be constantly adapted to the type of the data.

The compression level adaptation must be performed according to all these parameters at the same time. For instance, we have to take into consideration the ratio between the available bandwidth and the CPU power more than each criteria separately.

Table 1 shows compression timings. Two same size files have been compressed using either gzip [8] or lzf [13] tools on a 1 GHz PowerPC G4 under MacOS X 10.2.8. `oilpann.hb` is a sparse matrix file in the Harwell-Boeing format (ASCII). `bin.tar` is a tarball of executables. Lzf and gzip as well as their related libraries (liblzf and zlib) provide lossless compression based on the Ziv-Lempel algorithm [20, 21]. We see that lzf is a fast compression algorithm with low compression ratio. Concerning gzip, we see that the compression time (columns *c. time*) increases with the compression level as the decompression time (columns *d. time*) is roughly constant. After level 6 the compression ratio (columns *ratio*) does not increase significantly.

	`oilpann.hb`			`bin.tar`		
algo	c. time	ratio	d. time	c. time	ratio	d. time
lzf	1.5	3.26	2.7	2.3	1.68	3.2
gzip 1	4.4	4.88	2.7	8	2.23	3.1
gzip 2	4.4	5.13	3	8.6	2.27	3.3
gzip 3	4.6	5.52	3	10	2.31	3.1
gzip 4	6	5.83	2.5	11.5	2.38	2.9
gzip 5	6.6	6.32	2.9	12.3	2.43	3
gzip 6	8.1	6.64	2.5	16.3	2.44	3
gzip 7	10.1	6.75	2.8	18.4	2.45	3.5
gzip 8	26.7	6.99	3.8	24.1	2.45	3
gzip 9	46	7.02	2.6	34.3	2.46	3.2

Table 1. Compression Timings on Bench Files Using lzf and Different Levels of gzip

For some specific data it may happen that the size of the compressed data is larger than the size of the uncompressed data. This is the case for already compressed data. In this case, tools like gzip [8] guarantee that the size does not increase more than 0.0015% for such files.

In this paper, compression level 0 will mean no compression (no time is used to compress the data). For compression level 1 we will use lzf, for compression level 2 we will use gzip at level 1, ...

3. The AdOC Algorithm

3.1. Principle

The AdOC algorithm has been proposed by Jeannot, Knutsson and Bjorkman in [11]. It is a general-purpose user-level and portable algorithm suited not only for grid computing but also for any data transfer application. It is mainly based on two ideas:

- Compression and communication overlap. When a process performs some I/O (such as accessing a disk or a network socket) it is blocked until the device becomes ready. During that time, the processor is available to perform some computation. Overlapping compression with communication allows the compression time to become mostly invisible to the user. We also perform decompression and communication overlap on the receiver side for the same reason.

- Dynamic adaptation of the compression level. We saw in the previous section that the compression time depends on the compression level. Moreover, the environment (CPU/network speed, data, etc.) is subject to change with the time. Therefore, the available time to compress/decompress data changes during the data transfer. We adapt to the change of the environment by changing the compression level.

The AdOC algorithm is presented Figure 1, and works as follows. It uses:

- Multithreading. The sending process is made of two threads. One thread compresses the data. The other one sends the data on the network. On the receiving side the process is also made of two threads. One reads the network the other one decompresses the data. Multithreading allows to overlap the compression/decompression and the communication.

- FIFO queues. A queue is used to store data shared by the threads. On the sending side, the compression thread stores data in the queue, the emission thread reads this data and sends it to the network. On the receiving side, the reception thread reads the network and stores the data into the queue, the decompression threads reads the data from the queue to decompress it.

3.2. The compression thread

The compression thread has in charge to compress either a file or an array of bytes. In order to do that, it splits the file or the array into chucks of fixed size called *buffers*. The compression level is updated before reading a new buffer. Therefore, a tradeoff has to be found for the buffer size. If the size is too large, the reactivity needed to adapt the compression level may not be good enough. If the size is too small, the total amount of data sent will increase. Indeed, due to internal data structures of compression algorithms, compressing a file at a given level leads to a smaller compressed file than splitting the file, compressing each part and merging the compressed parts. In our implementation, the size of each buffer is chosen to be 200 KB. For this size, less than 6% of compression degradation is seen and the reactivity appears to be good enough [11].

Once the compression level is updated, a buffer is compressed at this level. Each time a *packet* of compressed data is generated, this packet is read and stored in the FIFO queue and the compression is resumed. In our implementation, the size of a packet is 8KB. If the compression is disabled, (compression level = 0) only an uncompressed packet is stored in the queue and the compression level is updated.

```
Input  n: number of packets in the queue
       δ: variation of the size of the queue
       l: old compression level
Output l: new compression level

1. if n=0
2.   return minLevel
3. if n< 10
4.   if δ ≤ 0
5.       l=l/2
6. else if n< 20
7.   if δ > 0
8.       l++;
9.   else if (δ < 0)
10.      l−−;
11. else if n< 30
12. if δ > 0
13.      l+=2;
14. else if δ < 0
15.      l−−
16. else if δ > 0
17. l+=2
18. l=max(l,minLevel)
19. l=min(l,maxLevel)
20. return l;
```

Figure 2. *Compression Level Update Algorithm*

3.3. Adapting the compression level with the FIFO queue

The AdOC algorithm monitors the size of the FIFO queue on the emission side as well as the variation of its size. The size of the queue is the number of stored packets. This information is used to update the compression level as shown in Fig. 2. The idea is the following:

- If the size of the queue increases, this means that the network and the receiver consume data slower than it is produced by the compression thread. Some extra time is therefore available for compression: the compression level is then increased.
- If the size of the queue decreases, this means that the network and the receiver consume data faster than it is produced by the compression thread. It is required to decrease the compression level in order to generate packets at a greater rate.

The goal of changing the compression level is to avoid the queue to become either empty or too large. If the queue

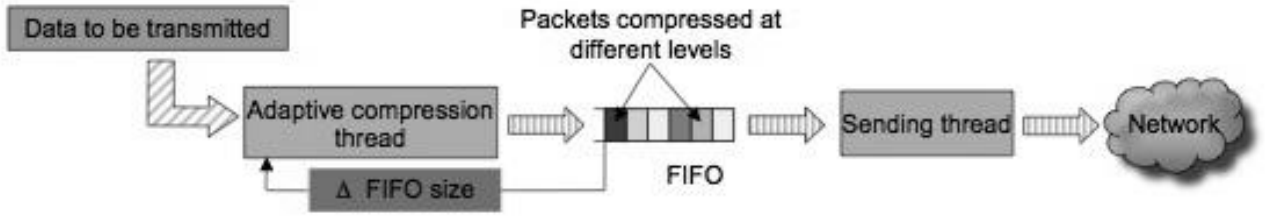

Figure 1. AdOC Algorithm: Emission Process (Reception Process is Symmetric but does not Monitor the Queue Size)

becomes empty, this means that the sending thread is waiting for data to be sent and therefore the transmission is slowed down. In order to avoid this to happen, some thresholds are added as describe in Figure 2. The compression level cannot increase if the queue size is too small (less than 10 packets). The level is increased by 2 (resp. divided by 2) if the queue is very large (resp. very small).

We see that the AdOC algorithm has a conservative strategy. As each packet has a size of 8 KB, and no compression is performed before the size of the FIFO becomes larger than 10 packets, no compression is done for data smaller than 80 KB.

4. AdOC Library

The AdOC library is an implementation of the AdOC algorithm. This library provides a set of user-level functions to send and receive data through sockets. The main features of this library are: synthetic API, full respect of the read/write UNIX system calls semantic, thread-safety, portability on many UNIX-like systems, efficiency on a broad range of networks (up to giga-ethernet LAN). Moreover, this library is available free of charge under the LGPL[1] license at `http://www.loria.fr/~ejeannot/adoc`.

4.1. AdOC library API

The AdOC library Application Programming Interface is very small and provides the ability to send and receive arrays of data or files. It also provides the ability to force or disable compression. The 7 functions of the API are the following:

- `ssize_t adoc_send_file(int d, FILE *pf, ssize_t *slen)`. This function sends the file pointed by `pf` to the object referenced by the descriptor `d` (a socket for instance). After the call, the number of sent bytes is pointed by `slen`. The size of the file is returned by the function. The compression ratio is therefore the ratio between the value returned by the function and the value pointed by `slen`.
- `ssize_t adoc_send_file_levels(int d, FILE *pf, ssize_t *slen, unsigned int min, unsigned int max)`. This function is the same as above, except that `min` sets the minimum level of compression to be used and `max` sets the maximum level of compression to be used. Two internal constants `ADOC_MIN_LEVEL` and `ADOC_MAX_LEVEL` define the minimum and maximum values for `min` and `max`. For instance setting `max` to `ADOC_MIN_LEVEL`, disables the compression while setting `min` to `ADOC_MIN_LEVEL`$+1$, forces the compression.
- `ssize_t adoc_receive_file(int d, FILE *pf);`. It reads an AdOC stream from the object referenced by descriptor `d`, decompresses the data if necessary and stores the data into the file pointed by `pf`. The amount of data stored is returned by the function.
- `ssize_t adoc_write(int d, void *buf, size_t nbytes, ssize_t *slen)`. This function is the same as the `write` UNIX system call except that the number of sent bytes is output in the `slen` pointer (it can be set to `NULL` if not used by the application). It writes the data pointed by `buf` to the object referenced by the descriptor `d`. The maximum number of data to write is given by `nbytes`.The function returns `nbytes` on success (a negative value in case of failure). Thanks to compression, the number pointed by `slen` must be lower than `nbytes`.

1 `http://www.gnu.org/copyleft/lesser.html`

- `ssize_t adoc_write_levels(int d, void *buf, size_t nbytes, ssize_t *slen, unsigned int min, unsigned int max)`. This function is the same as above with the ability to force or disable compression.
- `ssize_t adoc_read(int d, void *buf,size_t nbytes);`. This function is the same as the `write` UNIX system call. It reads an AdOC stream from the object referenced by descriptor `d` and stores the uncompressed data into `buf`. The maximum number of bytes to read is given by `nbytes`. The actual number of bytes read is return by this function.
- `int adoc_close(int d)`. This function is used to close the descriptor file `d` and to free AdOC internal buffers. In order to respect the `read`/`write` system call semantic it is required to be able to perform partial read. For instance a sender can send 100 MB, and the receiver can perform two `read`s one of 60 MB and one of 40 MB. In this case, temporary buffers are allocated to store received data. If the socket is closed after a partial read, temporary buffers have to be freed.

The ability of AdOC to send files is provided to ease the use of the library when files are to be sent. It is not intended to be competitive to the `sendfile` system call provided by some UNIX systems (such as LINUX). The main reason is that the `sendfile` system call does the file copy inside the kernel whereas AdOC is a user level library. Only `adoc_read`, `adoc_write` and `adoc_close` are intended to be used instead of the corresponding system calls.

4.2. Thread safety

The library does not use any global variable. A static variable is used to store and retrieve internal buffers when performing partial read. This variable is always accessed between locks. Therefore, different threads can use AdOC at the same time[2].

4.3. Portability

This library has been ported and compiled on many UNIX-like systems. It incorporates the compression library required by the algorithm (zlib [10] and liblzf [13]). So far AdOC has been successfully compiled and tested on the following platform/architectures: Linux, Solaris/SunOS, Darwin/MacOS, freeBSD, IBM AIX, SGI IRIX, dec-alpha OSF, cygwin as well as 64 bits linux kernels. We also ported the AdOC library to gcc/windows. However, tests show that cygwin outperform the gcc/windows version in most of the cases. Therefore, due to the difficulty to maintain two versions we provide only the cygwin one.

Note that, since we use the liblzf and the zlib, the compression is lossless, and therefore no alteration of the data are seen by the user.

5. Performance issues

In Section 3, we described an overview of the AdOC algorithm. We discuss here some performance issues that we have dealt with. This requires to change the algorithm in order the library to be efficient in broad range of scenarii.

Fast Networks In order the AdOC library to be general, one should not see performance degradation on fast Network. For some networks (up to 100 Mbit LAN), we need fast compression libraries that are able to compress the data to a speed at least equal to that of the network. We use the LZF library of Marc Alexander Lehmann [13]. As shown in Table 1, it is a very fast compression library that has about the same speed as the `memcpy` function[3]. The drawback of this library is that the compression ratio is very low (less than 2) therefore, we use this library as the first compression level (the second compression level corresponds to gzip at level 1).

Furthermore, very fast networks such as Gbit LAN are too fast for modern processors to have time to compress data even with lzf. In order to avoid performance degradation for such networks, we incorporate a bandwidth measurement into the protocol as follow. If the size of the data to transmit is large enough (512 KB) we measure the time to transmit a part of the data (256 KB) without compression. We deduce the speed of the link. If this speed is above 500 Mb/s, it means that we are dealing with a very fast network and we send the remaining data uncompressed, otherwise we use the adaptive algorithm.

The drawback of this approach is that no compression is performed if the size of the data is less than 512 KB whatever the network is. We think that this is reasonable as we target mainly large data set transfers and that 512 KB is less than the half of a 3.25 inches floppy disk capacity.

Compression level divergence The goal of the AdOC algorithm is to maintain the emission queue size to a reasonable value. If the queue size is empty, this means that no packets are sent to the network. If it is too large, this means that we have time to compress the data. However, when the receiver is very slow with regard to the sender, the adaptation process may diverge. Indeed, if we start compressing the data, the receiver will take a longer time to decompress it.

2 We have incorporated AdOC into the Internet Backplane Protocol (IBP) [4] that use multiple threads to store or retrieve data from data handlers. It works without error.

3 We could have used lzo [14], which has comparable performance to lzf, but its license is incompatible with the AdOC one.

Usually the compression time is far longer than the decompression time because it requires more computation power, but this is no longer true when both ends are very heterogeneous. If the compression time becomes smaller than the decompression time and the network is fast enough, the queue size will increase leading to an increase of the compression level. This is not the good choice, because the receiver will still be the bottleneck, the queue size will increase again leading to an increase of the compression level, etc. The good choice would be to disable the compression in such case.

The problem is that we want the library to respect the read/write semantic. Therefore, it is not possible for the receiver to send any information to the sender and ask it to stop the compression. Hence, the sender has to guess that the receiver is too slow for the compressed data it is sending. In order to solve this problem, we propose the following conservative strategy. The compression thread continuously measure the visible bandwidth and records it for each compression level. When updating the compression level, AdOC checks if the current level gives a better visible bandwidth than any smaller compression levels. If this is not the case this means that maybe we are facing a compression level divergence (an other reason could be that the network is temporally congested). Nevertheless, our conservative strategy gets back to the level that gives a better visible bandwidth and forbids the previous compression level for 1 second. After 1 second, we assume that the dynamic condition may have changed and we let AdOC try this level again if it decides it can be useful.

With this strategy the compression level is disabled when the receiver is not able to decompress data at a rate greater than its network arrival speed.

Small messages The AdOC library is a multithreaded library with a queue that is shared between the threads and accessed by mutexes. This adds some latency to the transfer. This latency has a cost that is visible for short messages on fast networks. Nevertheless, for small messages, compression is not very useful, and we measure the speed of the network by sending the first 256 KB uncompressed. Therefore, when messages are short (less than 512 KB), the data are sent uncompressed directly without launching the threads. In this case, the latency is the same than direct read and write calls.

Compressed and random data Some data, like random or already compressed one, takes time to be compressed and the obtained compression ratio is poor and sometimes smaller than one. In AdOC sending such data can lead to performance degradation. In order to avoid such a degradation we compare the size of each compressed packet to its original size. If the compression ratio is smaller than a given threshold, we stop compressing the remaining of the buffer and set the compression level to its minimal value for the next 10 packets before enabling compression again.

6. Experiments

The AdOC Library is designed to be used as a general-purpose communication service for any application instead standard POSIX read/write system calls. Hence, we have first measured its performance against those calls. Second, as it is intended to be incorporated into grid middlewares, we have plugged AdOC into the NetSolve [6] and compared application performance of both versions.

6.1. AdOC vs. POSIX read/write

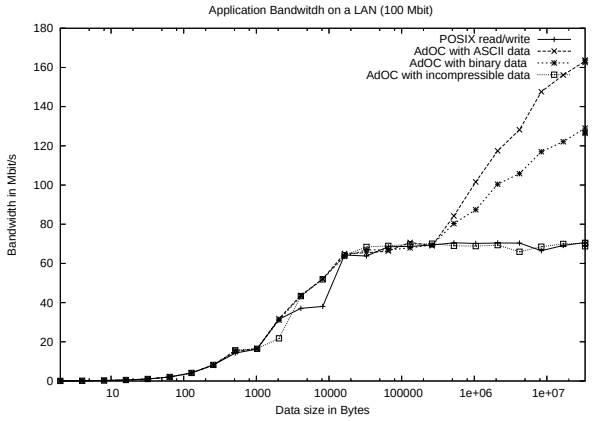

Figure 3. ***Bandwidth on a Fast Ethernet LAN***

6.1.1. Bandwidth Figures 3, 5, 4, 6 and 7 show the performance of AdOC compared to the POSIX read/write system calls. The experiments where performed using Linux machine, with 100 Mb network cards. On the x-axis, is shown the amount of transferred data in bytes. This axis use logarithmic scale. The sent data size is between 1 byte and 32 MB. On the y-axis, we show the bandwidth visible at the application level (by the user). It is evaluated by measuring the amount of time required by the application to send and received back a buffer of the given size.

Since the performance of AdOC depends on its capacity to compress data 4 drawings are shown on each figure. One represents the read/write performance. The three other drawings represent the AdOC timings with different data types. The first type represents ASCII data: it has a compression ratio of about 5 with gzip level 6. The second type represents binary data: it has a compression ratio of about 2 with gzip level 6. The last type represents incompressible

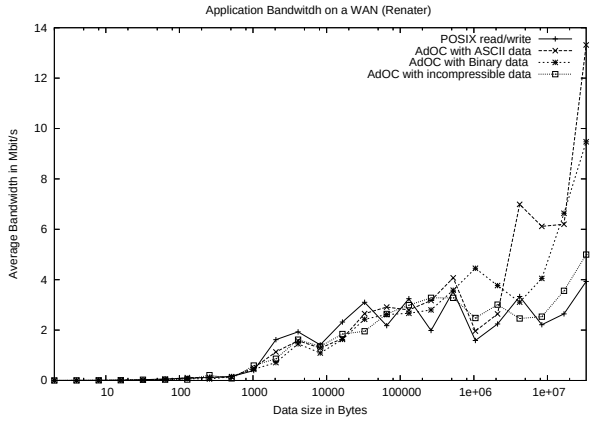

Figure 4. ***Bandwidth on Renater (Academic Network, between Nancy and Lyon), Average Timings***

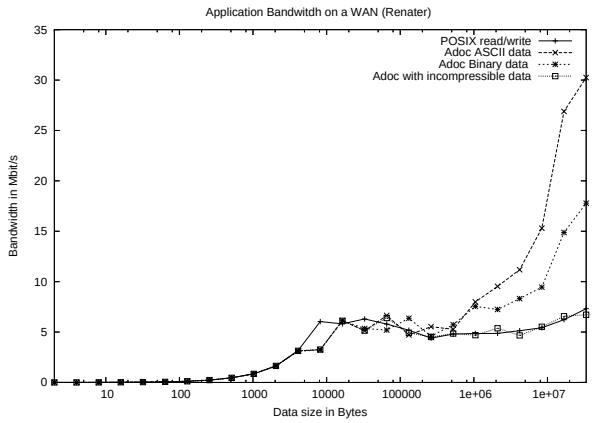

Figure 5. ***Bandwidth on Renater (Academic Network, between Nancy and Lyon), Best Timings***

data as gzip is not able to compress it. These data were generated randomly, the randomness being set accordingly to the desired compression ratio.

We believe that for most of the applications, data to be sent will be between the ASCII and the binary data.

Reproductivity of the experiments is a difficult issue. This is especially true on Internet and WAN where experiments are not reproducible. The standard deviation of the timings is very high and therefore it is difficult to conclude on the performance of each method based on average timings. To illustrate this phenomena, let us compare Figure 5 and Figure 4. On Fig. 4, each point represents the average

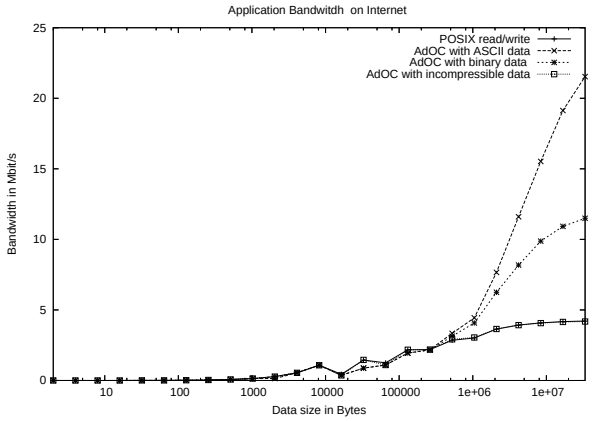

Figure 6. ***Bandwidth on Internet (Tennessee – France)***

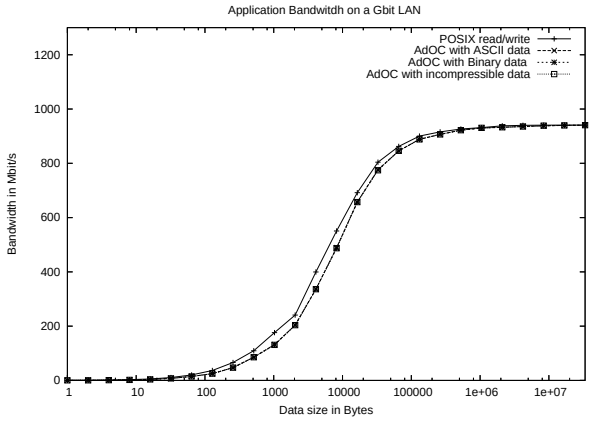

Figure 7. ***Bandwidth on a Gbit Ethernet LAN***

time of 40 measurements, on the Fig. 5, each point is the best time of 40 measurements. On one hand, we see that it is difficult to conclude on the behavior of each method with the plotting of average value (the average bandwidth is oscillating after 8 KB). On the other hand, plotting the best value gives smoother plots. Best-value plottings appear to be more reproducible. Therefore, we have decided to use only best values for *Renater* and *Internet* figures of this article. Another justification is that best value is fair for all the methods (with or without AdOC) : each of them is evaluated under the same circumstances: when network perturbation is minimal.

Results show that up to 512 KB, AdOC and POSIX read/write have the same performance: this is due to the fact that no compression is performed under this size. At that point and after, AdOC starts compressing data, and the time

	POSIX read/write	AdOC	AdOC with forced compression
Internet	80	80	225
Renater	9.2	9.2	25
100 Mbit LAN	0.18	0.20	1.8
Gbit LAN	0.030	0.045	1.6

Table 2. Latency of AdOC vs. POSIX read/write on Different Networks (in milliseconds)

to send data with AdOC becomes smaller than the time to send data with POSIX read/write. Not surprisingly the gain depends on the data and the network:

- On a 100 Mb Ethernet LAN (Fig. 3), for 32 MB, AdOC is between 1.85 and 2.36 times faster than the POSIX read/write.
- On Renater[4] between Nancy and Lyon (Fig. 4), for 32 MB, AdOC is between 6.1 and 2.6 times faster than the POSIX read/write.
- On Internet between France and Tennessee (Fig. 6), for 32 MB, AdOC is between 5.5 and 6 times faster than the POSIX read/write. The fact that the gain is smaller with Internet that with Renater is partially due to the fact that the machine we used in Tennessee was slower than the machines we used on Renater.

Moreover, we see that, for all these networks, for every size and type of data there is no performance degradation. Finally, the difference between AdOC with incompressible data and POSIX read/write is never significant: AdOC does not loose time with this kind of data.

For Gbit Ethernet (Fig. 7), we see a small degradation up to 1MB. This is the overhead of testing the network and the data size in order to guess if compression has to be used. However, in our tests the degradation does not depend on the size of the data: the overhead is between 10 and 20 μs.

6.1.2. Latency We have measured the average time of a zero byte ping-pong with AdOC and POSIX read/write. Results are shown in Table 2

We see that there is no difference between AdOC and POSIX read/write up to 100 Mb LAN. For Gbit LAN the latency is about 15 μs higher with AdOC. The latency given in the column AdOC with forced compression shows the overhead of starting the full AdOC process (threads, FIFO queue, mutexes, etc.) and the protocol overhead. Since these timings are very high, it justifies not to compress the data for small size.

4 Renater is the network that interconnects research center and university of france it provoides a backbone of several Gbit see `www.renater.fr`

6.2. AdOC into NetSolve

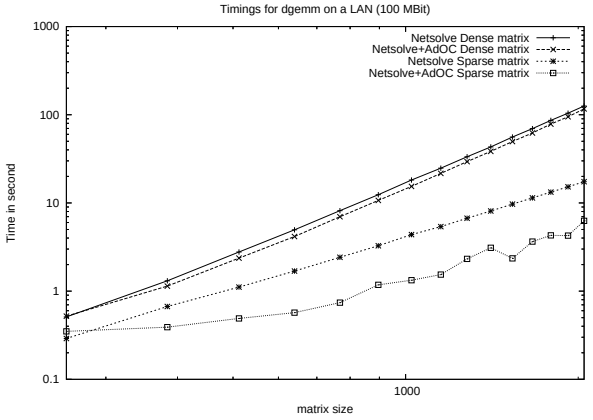

Figure 8. *NetSolve Timings on a 100 Mb LAN*

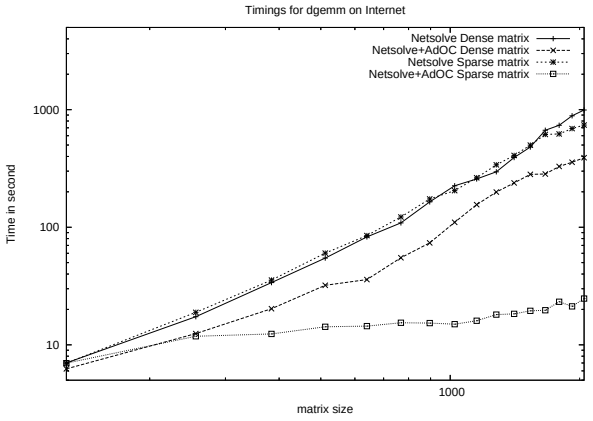

Figure 9. *NetSolve Timings on Internet (Tennessee-France)*

NetSolve is a middleware that works under the GridRPC model. It features a set of servers that register to an agent. When a client requests for a service it asks the agent to find the best suited server. It then executes the request to the server as a normal RPC.

We have modified NetSolve in order to enable AdOC in this middleware. This was very easy as it required to modify only the `communicator.c` file. We changed each

`read` call into `adoc_read` and each write call into `adoc_write`. We had also to change the makefiles so that NetSolve links against the AdOC library at compilation.

In Figures 8 and 9 we show the time to execute a dgemm[5] request on a LAN or on Internet using NetSolve. The agent and the server were on one end of the network whereas the client was on the other end. On the x-axis is shown the size of the matrix (number of lines or columns as matrices are square). On the y-axis we plot the time to perform the entire request. Each axis uses logarithmic scale. Two kinds of matrices where used. Matrix full of zero (called sparse matrix), matrix with 13 significant digits (as in some standard matrix libraries) and an exponent between 10^{-20} and 10^{+20} (called dense matrix). We do not use `oilpann.hb` file as it is a fix size ASCII matrix and we want to vary the size and use binary data. In our case a sparse matrix is very easy to compress : it is the best case. A dense matrix is hard to compress and should be considered as the worst realistic case.

For each kind of data the time with and without AdOC is plotted.

On a LAN (Fig. 8), we see that for dense matrices NetSolve with AdOC is slightly better than NetSolve without AdOC (5% faster for 2048*2048 matrix). On sparse matrix performance is better (up to 5.6 times faster for a 2048*2048 matrix). There is no performance degradation due to AdOC for any matrix size and any data type.

On Internet (Fig. 9), we see that NetSolve with AdOC always outperforms standard NetSolve. It is 2.6 times faster on a 2048*2048 dense matrix and 30.8 times faster on a 2048*2048 sparse matrix. We never see performance degradation due to AdOC on Internet too.

7. Related work

Several researches are done on using compression for transmitting data. In [12], the authors proposed an algorithm closed to the AdOC algorithm. They implemented this algorithm in the linux Kernel (TCP stack). Hence this implementation was not portable. With these authors we proposed the AdOC algorithm in [11].

In [18] the authors proposed a work close to ours. The adaptivity depends on the network, CPU and data. However, it ignores any related work on adaptive compression and this work is less general than ours as no library is provided and it does not work on 100 Mb LAN or higher. For high speed compression, it uses the Huffman algorithm that is slower and gives lower compression ratio than LZF.

In [19], an other adaptive compression study is performed. This is an ongoing work. This work highlights some problems of the original AdOC algorithm. These problems are all addressed in this paper. The compression is performed using threaded and non-threaded implementation. In the non-threaded implementation there is no overlap of communication and compression. It proposed a feedback mechanism in order to avoid compression level divergence. However, this mechanism requires to know the maximum available bandwidth of the network.

Compression to speedup data transfer is used in [2]. In this work the authors propose a Grid-enable computational framework based on Cactus [3] and Globus [9]. However, the compression was not adaptive: once, the compression is set, it is not possible to disabled it.

In [7], the authors propose an integrated solution for wide area communication on grids called NetIbis. Many features are proposed in this work and they use AdOC for enhancing the communication performance.

8. Conclusion

Data transfer is a key feature for computational and data grids. Such grids have to rely on efficient data transmission services that are able to provide fast transfer rate. Compression is one mean to increase the bandwidth see at the application level. However, the heterogeneous and dynamic nature of the grids required to adapt the compression to the environment.

In this paper we have presented the AdOC library. This library provides adaptive online compression for transferring data. The main features of the AdOC library are:

- The compression level is adapted according to the environment (current speed of the network and CPUs) and the data. The compression is lossless.
- It provides compression and communication overlap. AdOC is able to compress some part of the data while sending compressed or uncompressed packets.
- It works on a broad range of network (up to Gbit LAN)
- It is easy to incorporate into any existing software. AdOC is thread-safe and its API is very close to the read/write system calls and respects their semantics.
- It is ported on many UNIX like systems (LINUX (32/64 bits), SunOS, Darwin, Cygwin, etc.)
- It has a low latency: for small messages AdOC gives the same performance as POSIX read/write (up to 100 MBit LAN).

We have tested this library on various condition with various data types. First, it appears that there is almost no performance degradation due to AdOC (on Giga-Ehternet LAN, some microseconds are lost due to AdOC). Second, the performance gain obtained using AdOC depends on the data itself and the environment and can be very important (up to 6 times faster).

5 A dgemm is a matrix-multiplication program.

This library is intended to be used in any middleware that performs data transfer. We have incorporated the library into NetSolve. This was done easily thanks to the API close to the read and write system call. The performance of NetSolve with AdOC is never worst than NetSolve alone. Most of the results show an increase of performance for NetSolve with AdOC.

Our future work is directed towards extending the use of AdOC in existing software. An IBP data mover has already been proposed: it is needed to evaluate the performance precisely. The next software we target is gridFTP [1], where (as in FTP) a compression option is available.

We also direct our future work towards lossy compression for image transfer with various resolution. This is useful when a user has to choose one image among a set of images (thumbnails): the resolution and accuracy of the thumbnails is not necessary required to be very high.

9. Acknowledgments

We would like to thanks several people involved in the development of this library.

Aurelien Bouteiller from the LRI beta tested AdOC and fixed some bugs.

Ndoli-Guillaume Assielou, Bertrand Benoit and Alexandre Dombrat tested fast compression libraries such as LZF.

Hanane Moustain Billah worked on porting AdOC on the windows system (both natively and with Cygwin).

This work is partially supported by the french ACI GRID AGIR of the Ministry of Research.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group Document, 2002.

[2] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Conference on Supercomputing (SC 2001)*, Dever, Colorado, Nov. 2001.

[3] G. Allen, T. Goodale, and E. Seidel. The Cactus computational Collaboratory: Enabling Technologies for Relativistic Astrophysics, and a Toolkit for solving PDEs by Communities in science and engineering. In *7th IEEE Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, New-York, USA, 1999.

[4] A. Bassi, M. Beck, T. Moore, J. S. Plank, M. Swany, R. Wolski, and G. Fagg. The Internet Backplane Protocol: A Study in Resource Sharing. *Future Generation Computing Systems*, 19(4):551–561, May 2003.

[5] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, Aug. 2002. Springer-Verlag.

[6] H. Casanova and J. Dongarra. NetSolve : A Network Server for Solving Computational Science Problems. In *Proceedings of Super-Computing'96*, Pittsburg, 1996.

[7] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal. Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *Proceedings. of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 97 – 106, June 2004.

[8] P. Deutsch. *GZIP file format specification version 4.3*. IETF, Network Working Group, 1996. RFC1952.

[9] I. Foster and C. Kesselman. The Globus project: A progress report. In *Heterogeneous Computing Workshop*, Mar. 1998.

[10] J.-l. Gailly and M. Adler. zlib. `http://www.gzip.org/zlib/`.

[11] E. Jeannot, B. Knutsson, and M. Björkman. Adaptive Online Data Compression. In *IEEE High Performance Distributed Computing (HPDC'11)*, pages 379 – 388, Edinburgh, Scotland, July 2002.

[12] B. Knutsson and M. Björkman. Trading computation for communication by end-to-end compression. In *Third International Workshop on High Performance Protocol Architectures (HIPPARCH'97)*, 1997.

[13] M. A. Lehmann. liblzf. `http://www.goof.com/pcg/marc/liblzf.html`.

[14] M. F. Oberhumer. Lzo. `http://www.oberhumer.com/opensource/lzo/`.

[15] J. Postel and J. Reynolds. *FILE Transfer Protocol (FTP)*. IETF, Network Working Group, 1985. RFC 959.

[16] D. Rand. *The PPP Compression Control Protocol (CCP)*. IETF, Network Working Group, 1996. RFC1962.

[17] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H.Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture. In *HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[18] K. Schwan, P. Widener, and Y. Wiseman. Efficient End to End Data Exchange Using Configurable Compression. In *24th International Conference on Distributed Computing System (ICDCS 2004)*, Tokyo, Japan, Mar. 2004.

[19] L. Singaravelu, L. Kang, S. Park, and C. Pu. Effectiveness of Data Compression in Networks. Draft at `www.cc.gatech.edu/grads/l/lenin/docs/AC.ps`.

[20] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[21] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.