INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Adaptive Online Data Compression

Emmanuel Jeannot

N° 4400

Mars 2002

THÈME 1

Rapport de recherche

N 0249-6399

Adaptive Online Data Compression

Emmanuel Jeannot [*]

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n° 4400 — Mars 2002 — 11 pages

Abstract: Quickly transmitting huge data in the context of distributed computing on wide area network can be achieved by compressing data before transmission. However, such an approach is not efficient when dealing with high-speed networks. Indeed, the time to compress a large file and to send it is greater than the time to send the uncompressed file. In this paper, we propose an algorithm that allows to overlap communications with compression and to adapt the compression ratio according to the network speed (the slower the network, the more we use efficient and slow compression algorithms). The advantage of such an adaptive algorithm is its generality and that its suitability for a large set of applications.

Key-words: Distributed computing, efficient communications, adaptive network algorithm, online compression algorithm, communication overlap

[*] LORIA, Université H. Poincaré

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,

Compression adaptative et dynamique de données

Résumé : Pour transmettre des données de grandes tailles rapidement, dans le cadre du calcul distribué sur des réseaux à grande échelle, une des solutions possibles consiste à compresser ces données avant de les envoyer. Cependant, une telle approche ne fonctionne pas dans le cadre des réseaux rapides (100 Mbits/s) car le temps de compresser un grand fichier et de l'envoyer est supérieur au temps d'envoi du fichier non compressé. Nous présentons ici un algorithme qui permet de recouvrir la communication par la compression et d'adapter le taux de compression à la vitesse du réseau (plus le réseau est lent plus on utilise des algorithmes efficaces et lents pour compresser). L'avantage d'un protocole adaptatif de ce type est qu'il fonctionne bien pour un très grand nombre de réseaux et un large panel d'applications.

Mots-clés : Calcul distribué, communication efficaces, alagorithme adaptatif pour les réseaux, algorithme de compression dynamique, recouvrement de communication

1 Introduction

Recent developments in the area of grid-computing have focused on the need to efficiently transfer very large amount of data. Indeed, when designing Problem Solving Environments (PSE) such as NetSolve [3], Ninf [8], or Scilab$_{//}$ [2], data (files, internal objects, etc.) are to be sent. Performance of PSE greatly depends on the ability to efficiently transmit these data. However a meta-computing environment is composed of an heterogeneous set of machines interconnected by heterogeneous networks. Available computing resources can be old sequential machine as well as brand new parallel computers. These machines can be interconnected by a slow WAN (Internet), as well as a fast 2.5 Gbit/s backbone (VTHD[1]) or a 100 Mb-it/s LAN, etc.

Since no assumptions can be made on the hardware infrastructure we need a general transmission algorithm that is efficient whatever the speed of the network and the speed of the machines.

When the server is a parallel machine and the data to be transmitted are distributed among the nodes of the server one possibility to efficiently transfer these data is to use multi-socket technics. This solution has the advantage to try to use the maximum bandwidth of the backbone but is limited when a bottleneck exists (such has a slow network connected client) and is not possible when the server is a sequential machine. An other solution is to compress data prior to send them. However, this solution is inefficient in the case of a fast network when the time to compress the data and to send them is greater than the time to send the whole uncompressed data.

In this paper we propose an algorithm called *AdOC* (Adaptive Online Compression), which has the following features. First, it is a general algorithm suited, not only for grid computing, but for any data transfer (ftp, rcp, etc.) whatever the speed of the network. Second, it is a an adaptive algorithm which means the compression ratio depends on the amount of available time to compress (the slower the network the higher the compression ratio). Third, it is a dynamic algorithm with compression/communication overlap. This means that compression and communication are done simultaneously. Therefore, our algorithm is able to decide either to compress the data when useful or to send the data uncompressed when the network is fast.

Dynamic compression has been proposed for improving cache performance as shown in [4, 6]. But, as far as we know, there is no similar algorithm to AdOC proposed in the literature.

The remaining of the paper is organized as follow. Section 2 presents some important facts about communication and compression. The AdOC algorithm is proposed section 3. Section 4 presents our experimental results. Section 5 discusses possible optimizations of the algorithm. Concluding remarks are given section 6.

2 Compression and Communication

We present here some facts about compression and communication. We manly show that a none adaptive algorithm cannot be efficient for all kind of network and machines.

In the remaining of this paper we focus on the following problem. Given a file on a machine (the sender), how to transfer it as fast as possible to an other machine (the receiver).

2.1 Facts about Compression

A solution to efficiently transfer a file is to compress this file. However there is an overhead due to the compression time. We evaluate this overhead by measuring the time taken to compress and uncompress various files. We have taken two bench files. The first one is `oilpann.hb`. It is an sparse matrix in the Harwell/Boeing format (ASCII). This file has many redundancy and compresses very well. The second file is an archive of binaries (`bin.tar`), which compresses less than `oilpann.hb`. For sake of simplicity `bin.tar` have been truncated to the size of `oilpann.hb` (52 633 266 bytes).

Table 1 shows the time to compress the bench files using LINUX `gzip` with various compression options on a Pentium III 500MHz running LINUX kernel 2.4.7-10. `gzip` uses the Zempel-Liv compression algorithm [9] which basically checks for redundancy. It uses a "*sliding window*" which size defines the compression level (the larger the window the higher the compression level). Column *level* is the gzip compression option from 1 (fast method / low compression) to 9 (slow method / best compression). Column *c. time* is the LINUX user

[1] `http://www.vthd.org`

time to compress the file in seconds. Column *ratio* is the ration between the compressed size of the file and the uncompressed size. Column *d. time* is the LINUX user time to decompress the file. We remark that

	`oilpann.hb`			`bin.tar`		
level	c. time	ratio	d. time	c. time	ratio	d. time
1	10.2	4.88	9.6	16.9	2.23	12
2	11	5.13	9.3	17.2	2.27	12
3	11.3	5.52	8.9	18.3	2.31	13.4
4	11.9	5.83	9	21.7	2.38	13.7
5	13.6	6.32	8.7	23.6	2.43	13.6
6	16.7	6.64	8.3	28.3	2.44	13.7
7	20.4	6.75	8.7	32.1	2.45	13.14
8	47.6	6.99	7.7	44	2.45	13.1
9	76	7.02	7.8	65.8	2.46	12.3

Table 1: *Compression timings on bench files using gzip*

the compression time and ratio increase with the compression level. Moreover, for a given file, we remark that the decompression time is also roughly constant and is always smaller than the compression time.

2.2 Facts about Communication

One idea to transfer a file is to use FTP (file transfer protocol) or rcp (remote copy). Our experiments show that rcp and FTP have the same performances. Hence, we only show results for FTP. Table 2, shows FTP timings for transferring `oilpann.hb`[2] on various network type and various compression level. 100 Mbit/s experiments were done on the LAN of the LORIA whereas 10Mbit/s experiments were performed between two PCs interconnected by a cross cable. Transatlantic experiments were done between the LORIA (Nancy, France) and the university of Rutgers (New-Jersey, USA). Files were on the sender local disk and sent to the receiver local disk (NFS was not involved). Remark that the compression time are not included in the timing.

Row *FTP* is the time to send the file uncompressed. Row *level=1* (resp. *level=9*) is the time to send the file compressed with gzip using compression level 1 (resp. 9).

A second idea consists in using the `-C` option of the UNIX `scp` command that does a remote copy of a file using encryption and compression. The compression algorithm used by scp is the same used by gzip. The last two rows of table 2 show the timings of transferring the file using scp.

method	100 Mbit/s	10 Mbit/s	Transatlantic	ADSL	56k Modem
FTP	6.34	70.5	234	831	9900
level=1	1.08	14.5	50	171	2017
level=9	0.93	9.5	42.2	121	1452
`scp`	12	70	560	1010	10550
`scp -C`	23	68	220	342	1660

Table 2: *FTP and scp average timings on various network*

We can see that FTP/rcp is a good solution for high speed network. Indeed, we need about one second to transfer the compressed file but from 10 to 76 seconds to compress it and about 8 seconds to decompress it. Therefore sending the uncompress file is faster than compressing sending and then uncompressing the file.

[2]We have taken this file which compresses the best in order to minimize the transfer time

This is no longer true when dealing with slower network. In that case the dominant time is the transfer time and the compression/decompression time becomes negligible.

We also see that using scp is never a good idea, mostly because there is a high overhead due to encryption.

Thus, we can conclude that there is no general solution to efficiently transfer a file for any kind of network.

3 AdOC Algorithm

3.1 Overview

We have designed an algorithm that is able to decide what is the fastest way to transmit the file. Our current implementation is based on the TCP socket. It uses the two following features:

1. *Multi-threading*. The sender is composed of two threads. One thread (called the compression thread) is in charge of reading the file and compressing data when useful. An other thread (called the communication thread) is in charge of sending data. Such an architecture allows compression/communication overlap.
2. *FIFO*. A FIFO data structure is used to store data to be sent. The compression thread writes data to the FIFO. The communication thread reads the FIFO.

The most important value of the AdOC algorithm is the size of the FIFO. If the FIFO size is not growing this means that the sending thread consumes data faster than they are produced by the compression thread. In that case we deduce that we are dealing with a fast network and we have no time to compress data. On the other hand if the FIFO is growing this means that we are dealing with a slow network and hence we have some time to compress data and we can increase the compression level.

3.2 Compression/Communication Overlap

When a process/thread is doing some IO on a device (such as writing data to a file on a disk or to a socket on a network), it may be blocked waiting for the device to become ready. This especially happens when writing data to a socket because memory accesses are faster than network accesses. When this process/thread is blocked the processor is available for some computations. We use this system behavior in order to overlap communication with compression. The sending process is composed with two threads one for compression, one for communication. These two threads communicate using a FIFO data structure.

The compression thread produces data to be sent and the communication thread consumes these data and send them to the network. Such a pipeline schemes allow to hide the communication time with the compression time. Indeed, when the communication thread is suspended while sending data the processor is available for the compression thread to perform some computations.

Since the FIFO is a shared object between two threads we use a mutex and a semaphore to access it.

3.3 Compression Thread

The compression thread has in charge to read the file and to compress data if possible.

We use the *zlib* [7] from Gailly and Adler. It is a compression library that uses the same algorithm than gzip (Zempel-Liv). It provides a C API that is able, given a buffer and a compression level between 1 and 9, to compress this buffer in packets of a given size.

For instance, imagine we want to compress a buffer of 5 000 000 bytes with a packet size of 8192 bytes. If the compressed buffer has a size of 500 000 bytes we actually obtain 61 packets of 8192 bytes and one packet of 288 bytes. The nice thing with this API is that the compression function returns each time a packet is generated. The next call to the function resume the compression and generate the next packet, without any loss in the compression ratio. On the other side we can start decompressing the buffer each time a new packet arrives.

The algorithm of the compression thread is given Figure 1. fifo_size, is a global object that contains the number of non empty packets in the fifo. packet_size is a constant of the program. It is the maximum size of each packet stored in the fifo. When level = 0 (no compression is used), we read a packet from the file and store it in the fifo. buffer_size is also a constant of the program. It is the size of the buffer to be compressed (level≠0). A buffer is compressed into packets of size at most packet_size with the function compress_next_packet, which returns the compressed packet and its actual size. We add delimiters in the fifo to tell when a buffer is completely compressed or when the file is completely read. At the beginning of each step we save in prev_fifo_size the size of the fifo. At the end of each step we update the level of compression using the difference between the current size of the fifo and the size of the fifo at the beginning.

```
level=0;
while there is still data in the file
    prev_fifo_size=fifo_size;
    if (level=0)
        packet=readfile(packet_size);
        add_fifo(packet,packet_size,level);
        add_fifo(NULL,0,0);
    else
        buffer=readfile(buffer_size);
        while buffer is not fully compressed
            (packet,size)=compress_next_packet(buffer,packet_size,level);
            add_fifo(packet,size,level);
        add_fifo(NULL,0,0);
    level=update_level(level,fifo_size-prev_fifo_size);
add_fifo(NULL,0,0);
```

Figure 1: *Compression Thread Algorithm*

3.4 Updating Level of Compression

As shown Table 1 increasing the level of compression allows to decrease the size of the compressed file but increases the compression time. At the beginning we set the level to 0 (no compression) and we slowly increase this level if we are sure that we have enough time to compress buffers. Updating the level of compression works in a *slow start* conservative fashion : increasing the level is slow and decreasing the level is fast.

Figure 2 describes the algorithm for updating the level of compression. It uses the difference D between the size of the fifo at the end of a compression step and the size of the fifo at the beginning of the step. We have three cases :

1. D=0. This means that the number of packets produced by the compression thread is the same that the number of packets consumed by the communication thread.Hence we do not change the level of compression.

2. if D>0 this means that we are dealing with a slow network, hence we increase the level by 1 if the fifo size is greater than 10. We use this threshold in order to avoid the algorithm to switch from the non-compressing state to the compressing state too easily. This is particularly important in the case of a fast network.

3. D<0. The fifo size is decreasing and we divide the compression level by 2.

3.5 Tuning AdOC

We need to decide which value the buffer size and the packet size have to take.

```
input : level : current compression level, D : fifo variation size
output : new compression level.

if (D<0)
    level=level/2;
else if ((D>0) and (fifo_size > 10) and (level <9))
    level++;
```

Figure 2: *Updating the level of compression*

3.5.1 Determining the Buffer Size

Our algorithm split the file into buffers and compresses them independently. Due to the compression algorithm, the obtained compression ratio depends on the size of the given buffer (the larger the buffer, the better the compression ratio). However, since the compression level cannot be change while compressing a buffer, too large buffers have a bad impact on the reactivity of our algorithm. Hence, we need to find a trade-off for this value. Figure 3 shows the evolution of the compressed file size when increasing the buffer size for a level 1 compression of `oilpann.hb`. We see that the size rapidly decreases at the beginning and tends to a limit which is the compressed file size obtained using gzip. With gzip there is no buffer: the file is read and compressed in one step. We obtain similar results for `bin.tar` and various compression level. We have decided to choose a buffer size of 204 800 bytes because in most of the cases, with this size we obtain a difference less than 5% between the compressed file and the optimal compressed file size (the one obtained by gzip).

3.5.2 Determining the Packet Size

When not compressing the file (at the beginning of the algorithm or when dealing with a fast network), the file is splited into packets which are sent uncompressed. If we have too large packets the reactivity of our algorithm at the beginning will be affected. However if we have small packets, we will have a lot of elements added in the fifo which implies more overhead for managing the fifo and sending this packet. We need also here to find a trade-off. In order to have an idea of what is a good packet size we have performed the following computation which is inspired from optimizing the packet size for computation/communication overlap [5].

Let ν the packet size. Let B the size of the buffer to be compressed. Let ρ the compression ratio (hence we obtain a compressed buffer size $B' = B/\rho$). Let C the time to compress one byte (the buffer is compressed in time CB), and C' the time to uncompress one byte (the buffer is uncompressed in time $C'B$). Let γ_c the constant overhead to store a packet in the fifo. Let $t = \frac{B}{\rho\nu}$ the number of packets generated by the compression thread.

Hence the time to generate one packet is $T_a(B,\nu) = C\frac{B}{t} + \gamma_c = C\rho\nu + \gamma_c$, on the over hand the time to uncompress one packet is $T_c(B,\nu) = C'\rho\nu$.

For modeling the network we use a simplistic model that assumes a constant throughput and latency. Let γ_e the constant overhead to extract a packet from the fifo. Let τ the time to transmit one byte and β the network latency. Hence the time to send one packet is $T_b(\nu) = \gamma_e + \beta + \tau\nu$. This is an old and simplistic model but, as shown in the following, it will give us an idea of the order of the packet size.

We have a pipeline scheme, hence, the time to transmit a the whole buffer is :

$$\begin{aligned} T_t(B,\nu) &= T_a(B,\nu) + tT_b(\nu) + T_c(B,\nu) \\ &= C\rho\nu + \gamma_c + \frac{B}{\rho\nu}(\gamma_e + \beta + \nu\tau) + C'\rho\nu \\ &= (C + C')\rho\nu + \frac{B}{\rho\nu}(\gamma_e + \beta) + \gamma_c + \frac{\tau B}{\rho} \end{aligned}$$

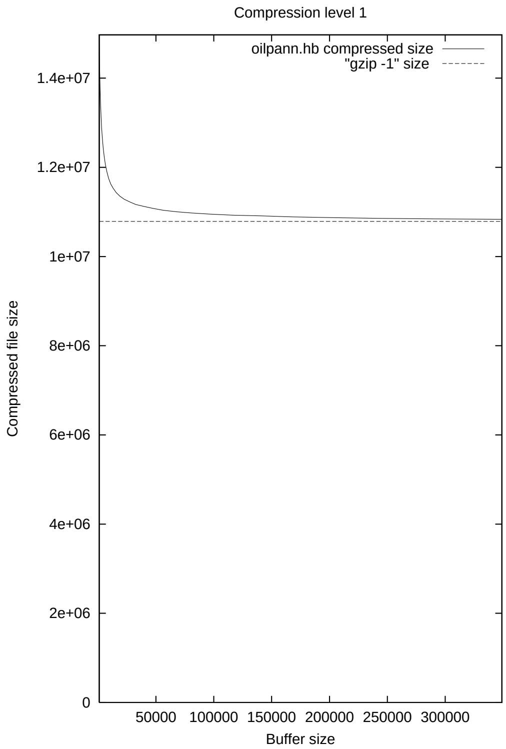

Figure 3: *oilpann.hb compressed size vs. buffer size*

We want to minimize $T_t(B,\nu)$. We have:

$$\frac{\partial T_t}{\partial \nu} = (C + C')\rho - \frac{B}{\rho}(\gamma_e + \beta)\frac{1}{\nu^2}$$

and

$$\frac{\partial^2 T_t}{\partial \nu^2} = 2\frac{B}{\rho}(\gamma_e + \beta)\frac{1}{\nu^3}$$

Since, $0 < \nu \leq \frac{B}{\rho}$ we have $\frac{\partial^2 T_t}{\partial \nu^2} > 0$. Hence $T_t(B,\nu)$ is concave and is minimum if

$$\frac{\partial T_t}{\partial \nu} = 0$$

This is true when

$$(C + C')\rho = \frac{B}{\rho}(\gamma_e + \beta)\frac{1}{\nu^2}$$

or when

$$\nu = \sqrt{\frac{B(\gamma_e + \beta)}{\rho^2(C + C')}}$$

Our experiments show that

$$1 \leq \rho \leq 7$$
$$100\,10^{-6}s \leq \beta \leq 50\,10^{-3}s$$
$$\gamma_e \approx 3\,10^{-6}s$$

$$1.9\,10^{-7}\text{s/byte} \leq C \leq 1.4\,10^{-6}\text{s/byte}$$
$$1.5\,10^{-7}\text{s/byte} \leq C' \leq 4.4\,10^{-7}\text{s/byte}$$

Since $B = 204800$ bytes we have $500 \leq \nu \leq 173\,550$. This interval is very large and just tell us that ν must be some kilobytes. Moreover, we want our algorithm to be very reactive at the beginning when the compression level is zero. Hence we wont take a large value for ν. Furthermore, a value of ν should be greater than a page size (4096 bytes). Finally, we have chosen the `BUFSIZE` constant of `stdio.h` for the packet size. It is also used in the FTP code for sending data and worths 8192 bytes on our system.

3.6 Communication Protocol

We need an application level protocol in order to correctly send the compress and uncompress packets.

At the beginning we send the buffer_size value. It allows the receiver to allocate the memory for the compressed buffer and for the uncompressed buffer. Thus, each time the communication thread send a fifo element it sends the packet size, the packet (if its size is greater that 0), and the compression level. If the level is zero the receiver writes the packet into the remote file. If the level is not zero the receiver uncompresses the packet and store the uncompress data in an output buffer. When the size is zero the output buffer is flushed to the remote file. When two packets of size zero are received in a row this means that the file has completely been sent and the receiver stops.

4 Experimental Results

We have built a sender and a receiver that implement the AdOC algorithm. The current implementation uses POSIX threads, mutex and semaphore. Communications are done using TCP sockets.

We have transmitted our two bench files on various networks. Results for `oilpann.hb` (resp `bin.tar`) are shown Table 3 (resp. Table 4).

We recall the timings of FTP and scp, but for *level=1* and *level=9*, we have added the time to compress and uncompress the file.

Row *AdOC C. ratio* shows the average compression ratio: the size of the file divided by the number of bytes sent by the communication thread.

method	100 Mbit/s	10 Mbit/s	Transatlantic	ADSL	56k Modem
FTP	6.34	70.5	234	831	9900
level=1	20.9	34.3	69.8	192	2037
level=9	84.7	93.3	126	205	1536
`scp`	12	70	560	1010	10550
`scp -C`	23	68	220	342	1660
AdOC	5,51	19.78	36.62	125.28	1180
AdOC c. ratio	1.16	5.26	5.92	6.74	6.98

Table 3: *AdOC average timings for oilpann.hb*

Results show that our algorithm outperform any other method for all kind of network. It especially works very well for average speed networks which are those used in distributed computing.

The average compression ratio decreases with the speed of the network. This shows that AdOC adapts its level of compression to the network.

5 Future Improvements

We would improve the performance of the receiver by decomposing it in two threads : one for reading the network and one for uncompressing the packets. This will allow to increase the network throughput since it will no longer depends on the speed of the receiver.

method	100 Mbit/s	10 Mbit/s	Transatlantic	ADSL	56k Modem
FTP	6.34	70.5	234	831	9900
level=1	33.8	60.6	133.8	401	4468
level=9	80.8	106.9	173.2	416	4102
`scp`	12	70	560	1010	10550
`scp -C`	48	79	373	485	5030
AdOC	6.31	36.04	90.07	340.5	4038
AdOC c. ratio	1.05	2.36	2.41	2.43	2.45

Table 4: *AdOC average timings for bin.tar*

An other improvement would be to have a faster compression method that would intercalate between level 0 and level 1. In that case the overhead of compressing files in very fast network would be even more hidden. Furthermore this will allow to have a smoothly transition from the non-compressing state to the compressing state.

On the other hand we could add better and slower compression algorithms (such has the one used in `bzip2` that is available within the *libbzip2* library and that has the same features than the zlib. This will allow to improve performances of AdOC for slow network.

6 Concluding Remarks

In this paper we have shown that there is no general solution that uses current algorithms and tools to efficiently transmit general data on any kind of network.

Our contribution is a general algorithm for efficiently transmitting data on a any kind of network. This algorithm adapts the compression level to the speed of the network and overlaps compression with communication. When dealing with a fast network the compression level is low whereas when dealing with a slow network the compression level is high. AdOC algorithm is especially adapted for grid-computing but we believe that its generality makes it a good candidate for a large set of applications.

As far as we know, it does not exist any other algorithm of this kind.

Future works are directed toward improving performances of AdOC and integrating this algorithm in PSE environments such as NetSolve [3] or DIET [1].

We are currently developing an open source library that will be available free of charge (check the author web page for more informations `http://www.loria.fr/~ejeannot`).

References

[1] DIET (Distributed Interactive Engineering Toolbox). `http://www.ens-lyon.fr/~desprez/DIET/index.html`.

[2] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J.M. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard. Scilab to Scilab$_{//}$, the OURAGAN Project. *Parallel Computing*, 27(11), 2001.

[3] H. Casanova and J. Dongarra. Netsolve : A network server for solving computational science problems. In *Proceedings of Super-Computing'96*, Pittsburg, 1996.

[4] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*, 1999.

[5] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.

[6] F. Douglis. The compression cache: Using online compression to extend physical memory. In *proceedings of the Winter Technical Conference, USENIX Association*, pages 519 – 529, January 1993.

[7] M. Adler and J.-L. Gailly. ZLIB. `http://www.gzip.org/zlib`.

[8] 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 Infrastructure. In *HPCN'97 (LNCS-1225*, pages 491–502, 1997.

[9] A. Zempel and J. Liv. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, 31(3):337 – 343, 1977.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399