Optimisation Mechanisms for MPICH/Madeleine

Nathalie Furmento, Guillaume Mercier

▶ **To cite this version:**

Nathalie Furmento, Guillaume Mercier. Optimisation Mechanisms for MPICH/Madeleine. [Research Report] RT-0306, INRIA. 2005, pp.29. ⟨inria-00069874⟩

HAL Id: inria-00069874

https://inria.hal.science/inria-00069874v1

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimisation Mechanisms for MPICH/Madeleine

Nathalie Furmento and Guillaume Mercier

N° 0306

July 2005

Thème NUM

ISSN 0249-0803

Optimisation Mechanisms for MPICH/Madeleine

Nathalie Furmento and Guillaume Mercier

Thème NUM — Systèmes numériques
Projet Runtime

Rapport technique n° 0306 — July 2005 — 29 pages

Abstract: This report presents optimisations mechanisms within MPICH/*Madeleine*, the implementation of MPICH over *Madeleine*. These mechanisms aim to decrease the communication time of derived datatypes for which data is stored in noncontiguous memory areas. The report presents the mechanisms as well as some performance evaluation.

Key-words: MPI, *Madeleine*, Derived datatypes, Communication times.

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 72 92 59 ??

Mécanismes d'Optimisations pour MPICH/Madeleine

Résumé : Ce rapport présente des mécanismes d'optimisation qui ont été introduits dans MPICH/*Madeleine*, l'implémentation de MPICH au dessus de *Madeleine*. Ces mécanismes ont pour but de diminuer les temps de communications pour les types dérivés. Ces types représentent des données qui ne sont pas stockées dans des zones de mémoire contigües. Le rapport présente également des tests de performances de ces mécanismes d'optimisation sur différentes architectures.

Mots-clés : MPI, *Madeleine*, Types dérivés, Temps de communication.

Contents

1 Introduction **5**
1.1 Context of our Work 5
1.2 Our Optimisation Mechanisms 5

2 Optimisation Mechanisms **6**
2.1 The Index Datatype 6
2.2 The Vector Datatype 7
2.3 The Struct Datatype 8
2.4 General Comments 9

3 Implementation Details **9**

4 Performance Evaluation **10**
4.1 Benchmark for the Index and Vector Datatypes 11
4.1.1 Benchmark with Fixed Number of Blocks 11
4.1.2 Varying the Number of Blocks 12
4.2 Benchmark for the Struct Datatype 14

5 Discussion **14**

A Detailed Performance Results **17**

B Source Code for the Performance Evaluation Programs **25**
B.1 The Index Datatype 25
B.2 The Vector Datatype 26
B.3 The Struct Datatype 28

List of Figures

1 Representation of the MPI index datatype 6
2 Representation of the MPI vector datatype 7
3 Representation of the MPI struct datatype 8
4 Definition of the index datatype for the benchmarks 10
5 Definition of the vector datatype for the benchmarks 11
6 Definition of the struct datatype for the benchmarks 11
7 Optimisation gains for the index datatype 12
8 Optimisation gains for the vector datatype 12
9 Performance evaluation for the index datatype over TCP when varying number of blocks 13
10 Performance evaluation for the index datatype over MX when varying number of blocks 13

11 Performance evaluation for the index datatype over GM when varying number of blocks 14
12 Optimisation gains for the struct datatype 15
13 Performance evaluation for the index datatype over TCP 17
14 Performance evaluation for the index datatype over MX 18
15 Performance evaluation for the index datatype over GM 19
16 Performance evaluation for the vector datatype over TCP 20
17 Performance evaluation for the vector datatype over MX 21
18 Performance evaluation for the vector datatype over GM 22
19 Performance evaluation for the struct datatype over TCP 23
20 Performance evaluation for the struct datatype over MX 24

1 Introduction

MPICH/*Madeleine* is an implementation of the MPI (Message Passing Interface) communication standard. This standard is widely used in scientific parallel computing applications. This library is an adaptation of the well-known MPICH (MPI CHameleon) [GLDS96, GL96], which has the property to be relatively easily adaptable on top of new network technologies thanks to a well-defined layered architecture. This implementation of MPICH over the *Madeleine* III library [Aum02] was designed and realised by Guillaume Mercier during his PhD thesis [Mer04, AM03].

We aim to introduce optimisation mechanisms to deal with the transfer of data which is stored in noncontiguous memory areas.

1.1 Context of our Work

The basic MPI communication mechanisms can be used to send or receive a sequence of identical elements that are contiguous in memory. To send data that is not homogeneous or that is not contiguous in memory, application developpers can define a derived datatype to specify more general data layouts.

Four derived datatypes are available:

1. **The contiguous datatype: MPI_Type_contiguous**
   The simplest constructor. Produces a new datatype by making count copies of an existing data type.

2. **The vector datatype: MPI_Type_vector and MPI_Type_hvector**
   Similar to contiguous, but allows for regular gaps (stride) in the displacements. `MPI_Type_hvector` is identical to `MPI_Type_vector` except that the stride is specified in bytes.

3. **The index datatype: MPI_Type_indexed and MPI_Type_hindexed**
   An array of displacements of the input data type is provided as the map for the new data type. `MPI_Type_hindexed` is identical to `MPI_Type_indexed` except that offsets are specified in bytes.

4. **The struct datatype: MPI_Type_struct**
   The most general of all derived datatypes. The new data type is formed according to completely defined map of the component data types.

In the following of this report, we will not consider the datatype `MPI_Type_contiguous` as the data it represents is contiguous in memory.

1.2 Our Optimisation Mechanisms

When sending a message described with a derived datatype, MPICH will first pack the data of this message in a new buffer, and then send this buffer as a single contiguous memory area.

As *Madeleine* can send in the same single communication data which are not contiguous in memory, we propose to let *Madeleine* deal with this noncontiguous message. We expect the cost of *Madeleine* processing the different parts of the message will be much smaller than the initial MPI cost of allocating a new buffer and copying the different parts of the message into it.

A *Madeleine* message consists of several pieces of data, located anywhere in user-space. It is initiated with a call to the method `mad_begin_packing()`. Each data block is then appended to the message by calling the method `mad_pack()`. Eventually, the message construction is finalized by calling `mad_end_packing()`. We can see that this structured building-message mechanism is well-adapted to the definition of a MPI derived datatype, as each subpart of the data can be inserted in the message using the method `mad_pack()`.

2 Optimisation Mechanisms

This section presents the algorithms used by *Madeleine* to transfer data described by the MPI derived datatypes.

2.1 The Index Datatype

This derived datatype replicates a datatype, taking blocks at different offsets. It allows one to specify a noncontiguous data layout where displacements between successive blocks need not to be equal.

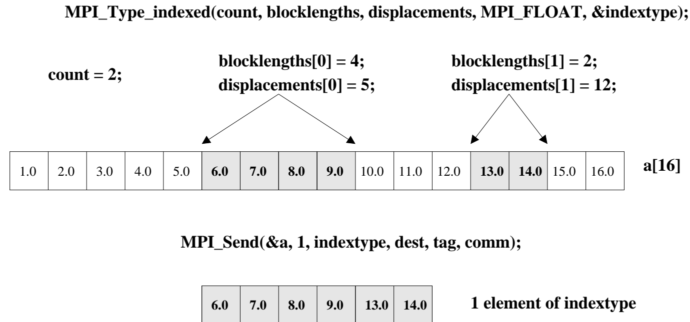


Figure 1: Representation of the MPI index datatype

From Figure 1, one can see each block can be sent with an unique call to the method `mad_pack()`. Additional informations such as the number of blocks and the number of elements in each block also need to be sent (see Algorithm 1).

Algorithm 1 Transfer algorithm for the index datatype

1: pack the number of blocks i.e. *count*
2: pack the size of an individual element e.g $sizeof(MPI_FLOAT)$
3: pack the number of elements in each block i.e. the array *blocklengths*
4: **for each** block b **do**
5: pack the data contained in the block i.e. $a[displacement[b] \ldots displacement[b] + blocklengths[b]]$
6: **end for**

The number of calls to the method `mad_pack()` is as follows:

- 2 calls for [1 * sizeof(integer)] (line 1 + line 2)
- 1 call for [number of blocks * sizeof(integer)] (line 3)
- number of blocks calls for [number of elements in the block * sizeof(element)] (line 4 * line 5)

2.2 The Vector Datatype

This derived datatype also consists of the replication of a datatype by defining regular gaps or overlaps (stride) in the displacements (see Figure 2).

MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);

count = 4; blocklength = 1; stride = 4;

1.0	**2.0**	3.0	4.0
5.0	**6.0**	7.0	8.0
9.0	**10.0**	11.0	12.0
13.0	**14.0**	15.0	16.0

a[4][4]

I=1 ; MPI_Send(&a[0][I], 1, indextype, dest, tag, comm);

2.0	**6.0**	**10.0**	**14.0**

1 element of columntype

Figure 2: Representation of the MPI vector datatype

As for the index datatype, one can see from Figure 2 that each block can be sent with an unique call to the method `mad_pack()`. The algorithm for the vector datatype slightly differs from the algorithm of the index datatype as for the vector datatype, the number of elements in each block is identical (see Algorithm 2).

Algorithm 2 Transfer algorithm for the vector datatype

1: pack the number of blocks i.e. $count$
2: pack the size of an individual element e.g. $sizeof(MPI_FLOAT)$
3: pack the number of elements in a block i.e. $blocklength$
4: **for each** block b **do**
5: pack the data contained in the block i.e. $a[I+b*stride \dots I+b*stride+blocklength]$
6: **end for**

The number of calls to the method `mad_pack()` is as follows:

- 3 calls for [1 * sizeof(integer)] (line 1 + line 2 + line 3)
- number of blocks calls for [number of elements in the block * sizeof(element)] (line 4 * line 5)

2.3 The Struct Datatype

This derived datatype is the most general one and allows to gather a mix of different datatypes scattered at many locations in space into one datatype (see Figure 3).

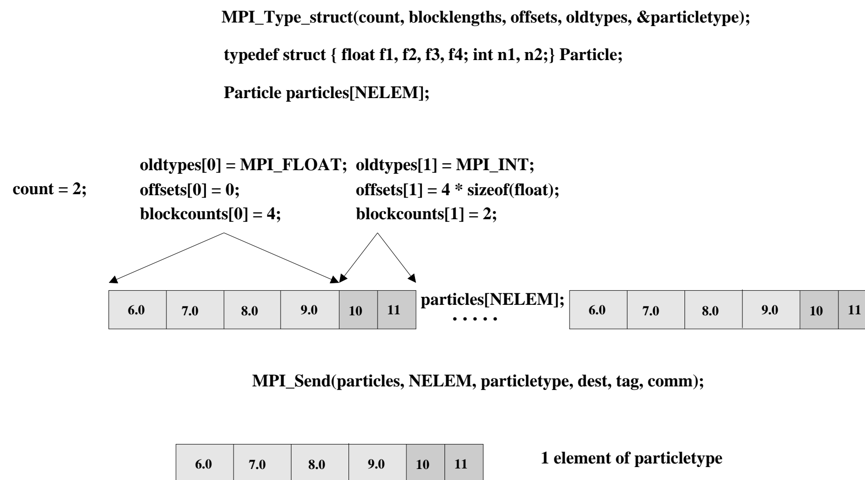


Figure 3: Representation of the MPI struct datatype

Being completely irregular, this datatype requires many information to be transfered (see Algorithm 3).

Algorithm 3 Transfer algorithm for the struct datatype

1: pack the number of elements i.e. $NELEM$
2: pack the number of blocks in each element i.e. $count$
3: pack the number of data for each block i.e. the array $blockcounts$
4: pack the displacement for each block i.e. the array $offsets$
5: pack the size of an individual data for each block i.e. the array $oldtypes$
6: **for each** element e **do**
7:   **for each** block b in the element e **do**
8:     pack the data contained in the block of this element i.e. $p[e][offsets[b] \ldots offsets[b] + blockcounts[b]]$
9:   **end for**
10: **end for**

The number of calls to the method `mad_pack()` is as follows:

- (2 + number of blocks) calls for [1 * sizeof(integer)] (line 1, line 2, line 5)
- 2 calls for [number of blocks * sizeof(integer)] (line 3, line 4)
- (number of elements * number of blocks) calls for [number of data in the block * sizeof(data)] (line 6 * line 7 * line 8)

The line 5 of the algorithm sends the size of the data. This information is not packed as an array of *count* integers, but packed as *count* integers. This is due to the fact that when the operation is performed, the information is no longer available as an array.

2.4 General Comments

We clearly see that the number of calls to the method `mad_pack()` depends on the definition of the derived datatype. And for the struct datatype, the number of elements is also a direct factor to the number of calls to the method `mad_pack()`.

A first assomption would be that the performance of our technique will be directly connected to the definition of the derived datatypes, and may decrease as the number of blocks in the datatype increases, and the size of the blocks decreases.

3 Implementation Details

The optimisation mechanisms are implemented within MPICH/*Madeleine* when the macro `DERIVED_DATATYPE_OPTIMIZED` is defined. That macro can be defined in the file `mpid/ch_mad/synchro.h`. We are in the process of adding a new functionality that will allow users to enable or disable the optimisation directly from their MPI applications by using attributes of communicator objects.

The main modifications to the source code were related to the transfer of the datatype information down to the level where the data is physically sent. A normal implementation of MPI does not need to get the datatype information down to the physical transmission level as it only needs a pointer to the data to be sent and the length of this data. These modifications required to update signatures of some of the MPI functions and to add the necessary code to transfer the datatype information down to the required level. This was easily done due to the layered structure of MPICH. Indeed all the modifications are centralised in the *Madeleine* device of MPICH/*Madeleine*.

Once the physical transmission level is reached, thanks to the datatype information, it is possible to send the data following the algorithms presented in Section 2. Symmetrical algorithms are applied on the receiving side to extract the information and to reconstruct the data to pass back to the application level.

4 Performance Evaluation

This section presents some performance evaluation of our optimisation mechanism. The program used is a ping-pong between 2 processors $P0$ and $P1$, we measure the time for the processor $P0$ to send data of a specific derived datatype to $P1$ and to receive that same data back from $P1$. The code of the method performing this ping-pong is available in Appendix B. The parameters of the ping-pong are s, the size of the message, and b, the number of blocks the data is decomposed in. Each test is repeated n times for each value of s and b, and the minimum time obtained for these 2 values is considered as the result of the experiment.

The machines used for the test are 2 Hyper-Threaded Bi-Pentium IV. They are connected with both a TCP network and a Myrinet network accessed via the MX protocol. We also used 2 Itanium 2 connected with a Myrinet network accessed with the GM protocol.

Index datatype. For that benchmark as shown on Figure 4, we send an array of n floats splitted in b blocks, the size of each block being n/b ; plus the last $n\%b$ elements for the last block.

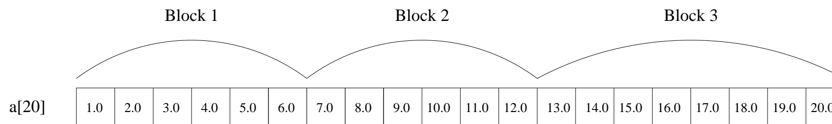


Figure 4: Definition of the index datatype for the benchmarks

Vector datatype. For that benchmark as shown on Figure 5, we send a matrice of $n * n$ floats splitted in b blocks, the size of each block being $n * n/b$.

a[4][4]
b = 4

1.0	2.0	3.0	4.0	Block 1
5.0	6.0	7.0	8.0	Block 2
9.0	10.0	11.0	12.0	Block 3
13.0	14.0	15.0	16.0	Block 4

Figure 5: Definition of the vector datatype for the benchmarks

Struct datatype. For that benchmark as shown on Figure 6, we send a structure including 2 float elements followed by 1 integer element followed by 1 float element. For that specific type, we fixed the number of blocks to 3. The blocks represent the three sets of data in the structure.

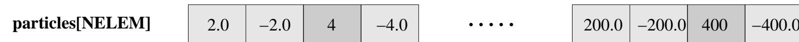


Figure 6: Definition of the struct datatype for the benchmarks

4.1 Benchmark for the Index and Vector Datatypes

This section presents the performance obtained for the index and vector datatype. Performance evaluation for the struct datatype is shown in Section 4.2.

4.1.1 Benchmark with Fixed Number of Blocks

In a first experiment, we fix b, the number of blocks to 3 and only increase s, the size of the message. Figures 7 and 8 show a global overview for the three types of networks of the gains obtained with our optimisation mechanisms for the index and vector datatypes. Figures 13, 14 and 15 give a detailed view for a specific type of network of the performance and the gain obtained for the index datatype. The detailed views for the vector datatype are shown on Figures 16, 17 and 18 in Appendix A.

One can see the performance for the vector datatype are negligible, but are good for the index datatype over the Myrinet network (reaching 23 %). The jump in performance for the index datatype over the GM protocol (falling down from about 12 % for 98 K floating elements to about 3 % for 99 K floating elements) is explained by the fact GM switches from an eager protocol to a rendezvous protocol to send the data. And the synchronisation involved in the rendezvous protocol affects the performance of our optimisation mechanism.

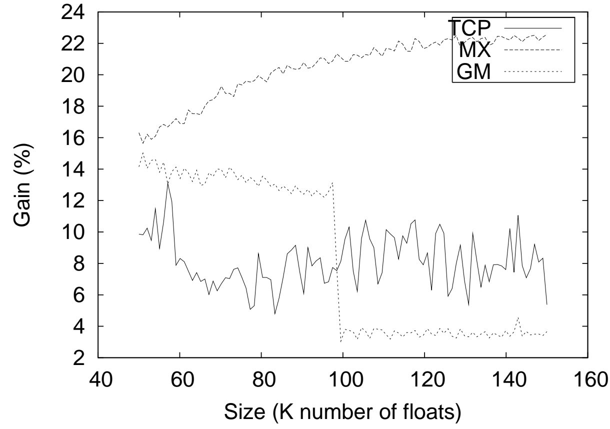


Figure 7: Optimisation gains for the index datatype

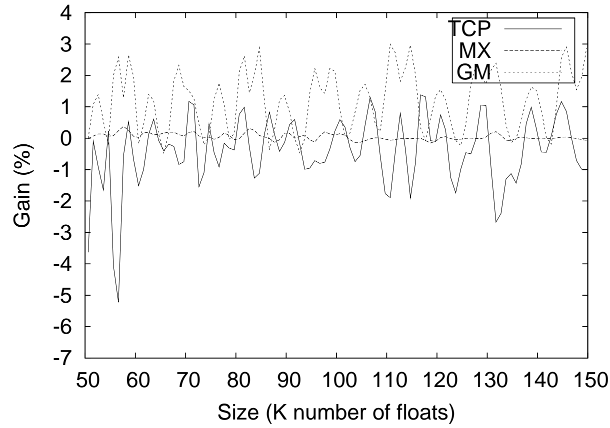


Figure 8: Optimisation gains for the vector datatype

4.1.2 Varying the Number of Blocks

For that experiment, we fix s the size of the message and vary b the number of blocks from 1 to 10. One can see from Figures 9, 10 and 11 that this factor does not have a big and regular effect on the optimisation gain. Performance certainly decreases when using MX for a subset of size of messages, but it also increases for another subset. Our first assumption that the number of calls to the method `mad_pack()` would affect highly the performance of

the optimisation mechanisms is not correct for the index datatype. However, we will see in Section 4.2 that it is the case for the struct datatype.

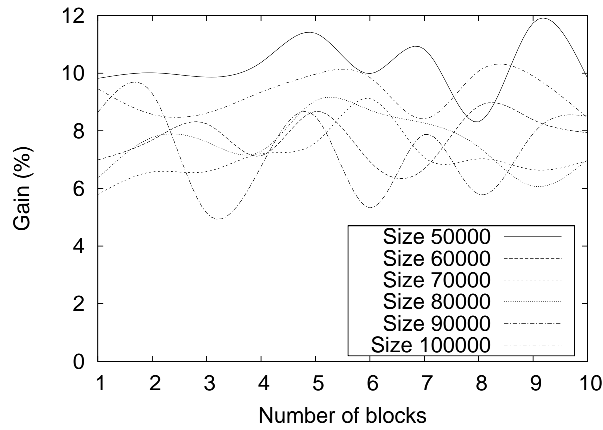


Figure 9: Performance evaluation for the index datatype over TCP when varying number of blocks

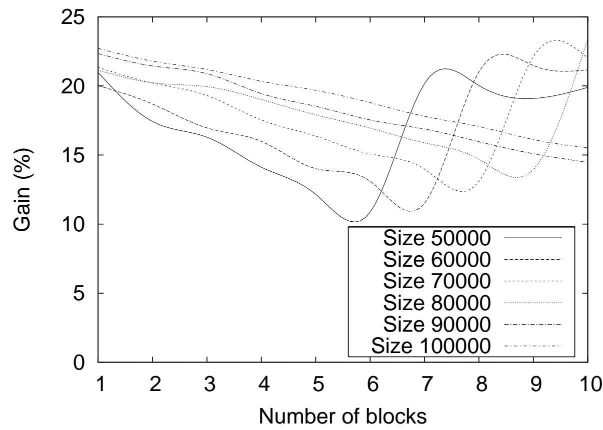


Figure 10: Performance evaluation for the index datatype over MX when varying number of blocks

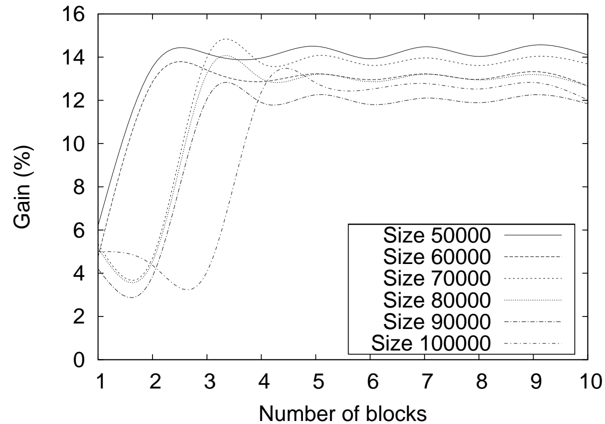


Figure 11: Performance evaluation for the index datatype over GM when varying number of blocks

4.2 Benchmark for the Struct Datatype

Figure 12 shows a global overview for the TCP network and the Myrinet/MX network of the gains obtained with our optimisation mechanisms for the struct datatype. Detailed views are shown in Figures 19 and 20 in Appendix A. Performances for that type are highly negative, this is due to the fact that the number of calls to the method `mad_pack()` is too high (it is a direct factor of the number of elements sent and the number of blocks in each element), and *Madeleine* is not able to exploit efficiently messages composed of too many packets.

5 Discussion

We can see that the performances are globally good for the index datatype when this datatype split the data in huge memory areas. Regarding the vector datatype, the performance gains are almost negligible. However, for the struct datatype, the new mechanism is not efficient as it ends up in a huge number of calls to the method `mad_pack()`, each call for a relatively small memory area.

These experiments have shown possible optimisations for *Madeleine*, when the number of calls to the method `mad_pack()` is large, *Madeleine* could obtain some gain by flushing the data when the size of the data already available is larger than a given threshold. That would lead to an overlap between the sending of some packets and the processing of the following ones. An other optimisation could be to consider an extended pack in which a reference to the whole data and the description of the datatype would be directly transfered

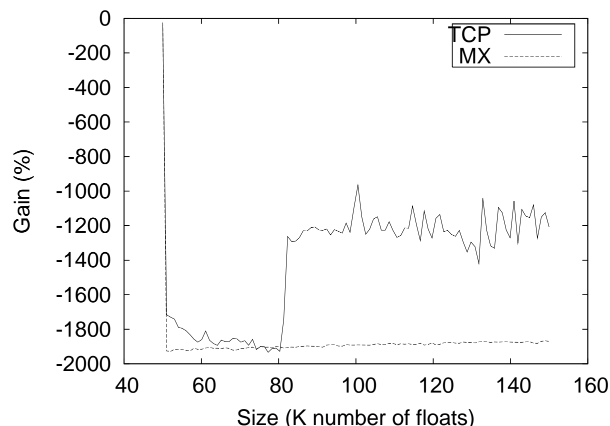


Figure 12: Optimisation gains for the struct datatype

down to the physical transmission layer. This layer would, for instance, get the information that each other n cells of a vector V has to be transfered, and would potentially be able to perform some pertinent optimisations.

References

[AM03] O. Aumage and G. Mercier. MPICH/MadIII: a Cluster of Clusters Enabled MPI Implementation. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 26–32, Tokyo, May 2003. `http://csdl.computer.org/comp/proceedings/ccgrid/2003/1919/00/1919toc.h%tm`.

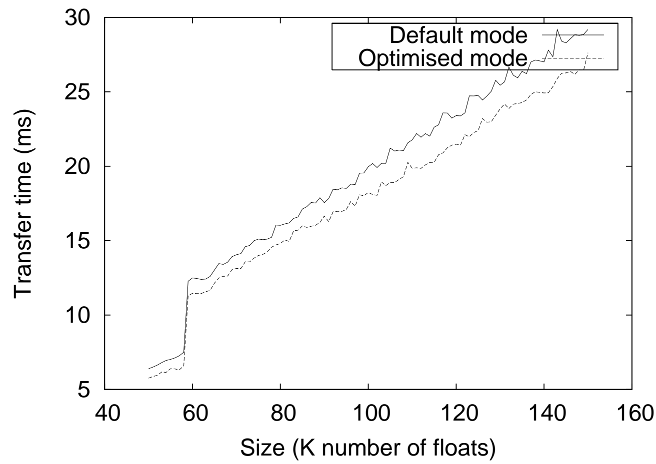
[Aum02] O. Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002. 12 pages. Extended proceedings in electronic form only.

[GL96] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

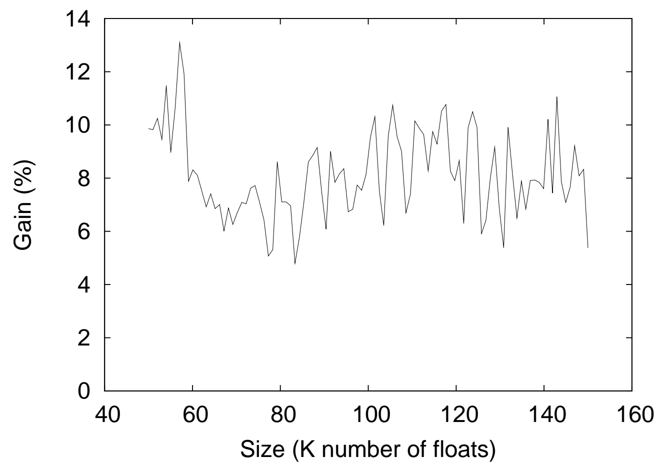
[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[Mer04] G. Mercier. *Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques.* PhD thesis, Université Bordeaux I, December 2004.

A Detailed Performance Results

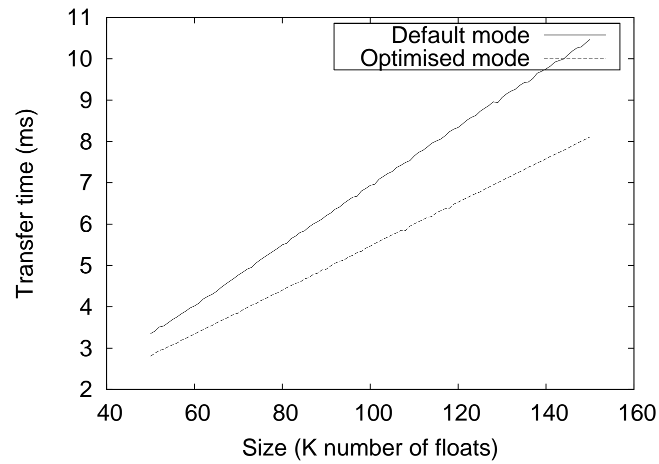


(a) Ping-pong time

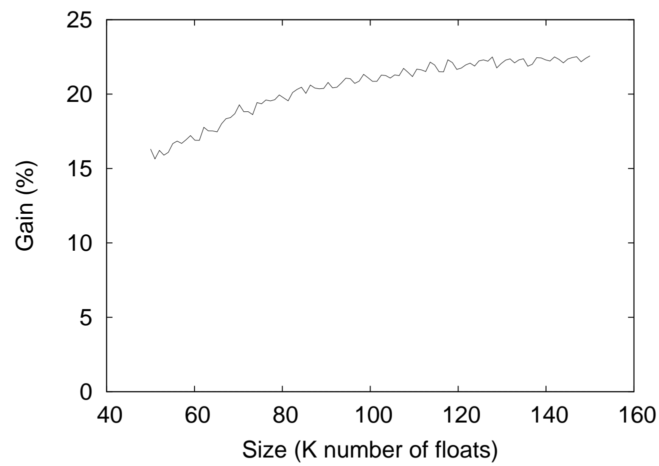


(b) Optimisation gain

Figure 13: Performance evaluation for the index datatype over TCP

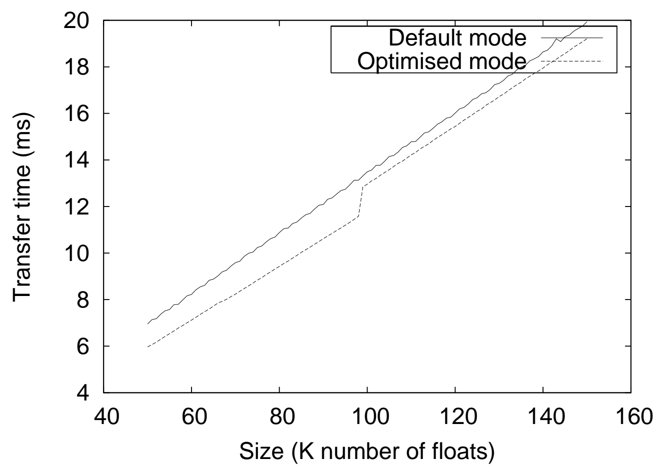


(a) Ping-pong time

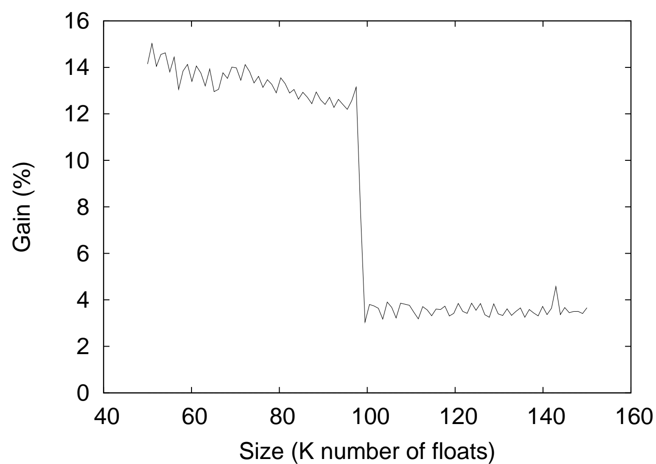


(b) Optimisation gain

Figure 14: Performance evaluation for the index datatype over MX

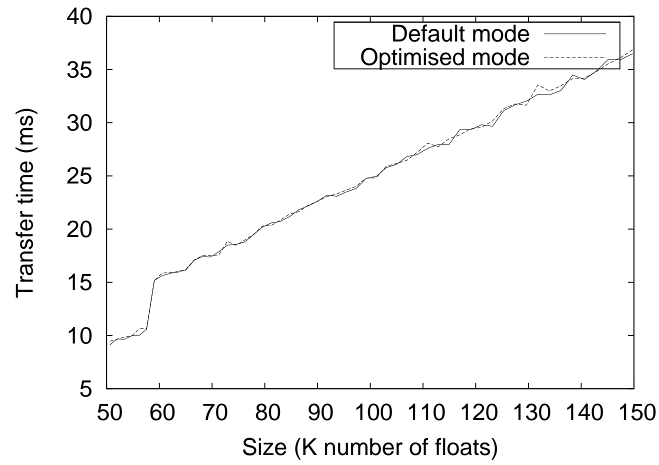


(a) Ping-pong time

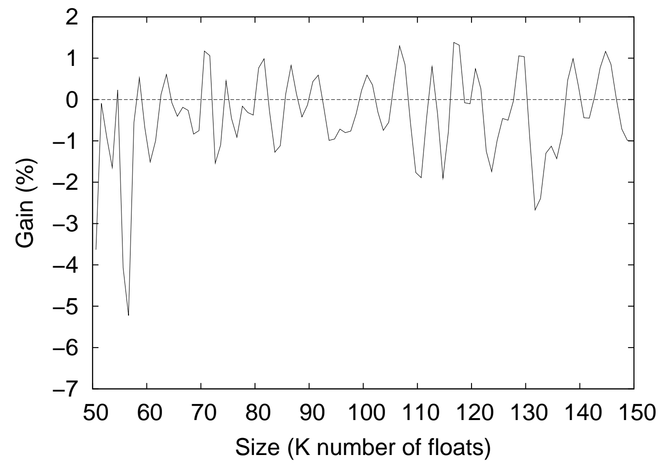


(b) Optimisation gain

Figure 15: Performance evaluation for the index datatype over GM

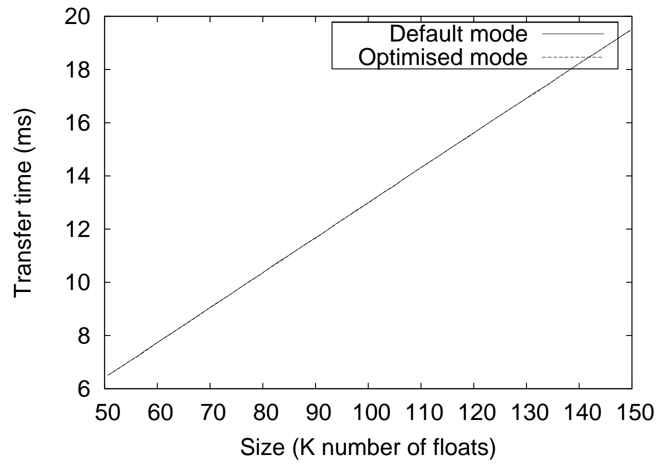


(a) Ping-pong time

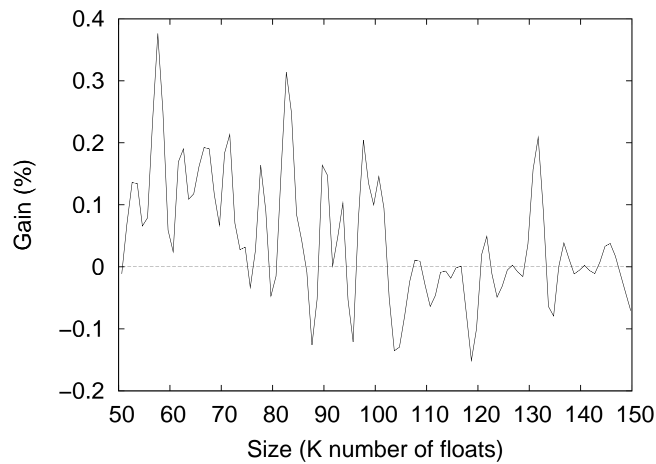


(b) Optimisation gain

Figure 16: Performance evaluation for the vector datatype over TCP

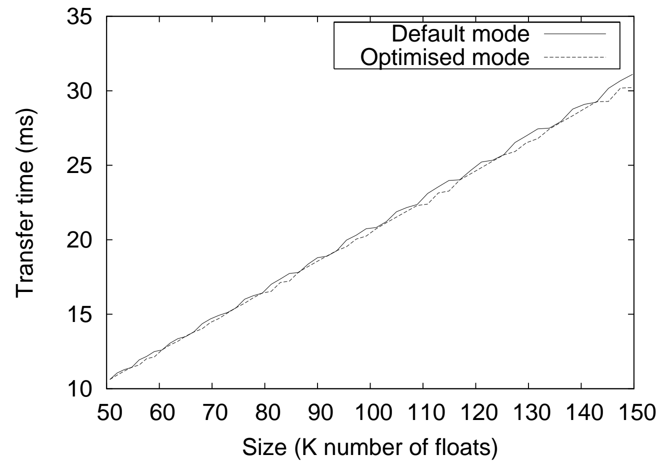


(a) Ping-pong time

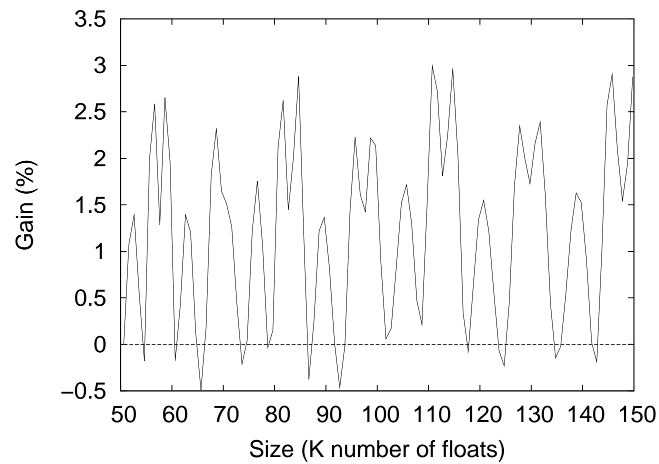


(b) Optimisation gain

Figure 17: Performance evaluation for the vector datatype over MX

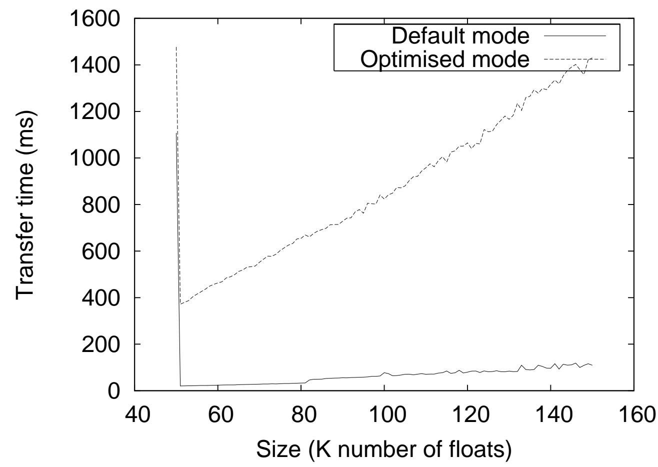


(a) Ping-pong time

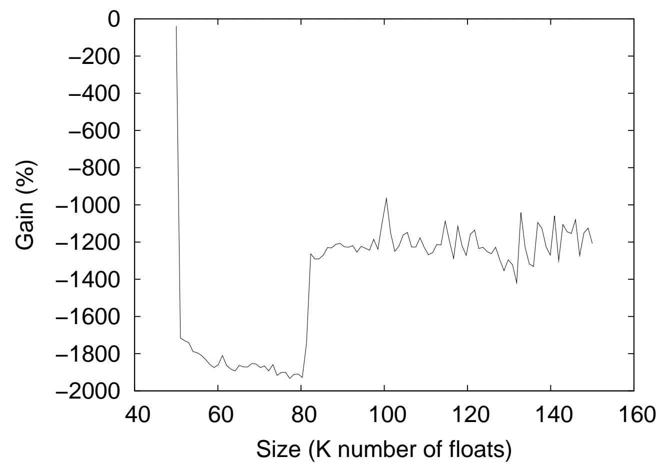


(b) Optimisation gain

Figure 18: Performance evaluation for the vector datatype over GM

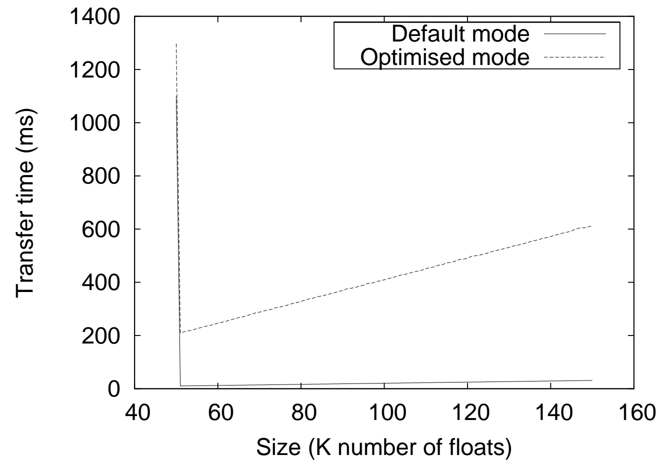


(a) Ping-pong time

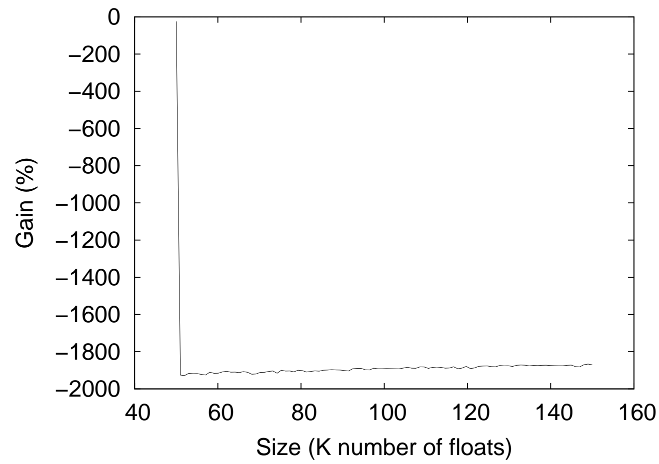


(b) Optimisation gain

Figure 19: Performance evaluation for the struct datatype over TCP



(a) Ping-pong time



(b) Optimisation gain

Figure 20: Performance evaluation for the struct datatype over MX

B Source Code for the Performance Evaluation Programs

B.1 The Index Datatype

```
void sendIndexTypeFromSrcToDest(int numberOfElements, int blocks, int rank, int source,
                                int dest, int numtasks, int use_hindex) {
  int          blocklengths[blocks];
  int          displacements[blocks];
  MPI_Datatype indextype;
  MPI_Status   stat;
  int          i;
  tbx_tick_t   t1;
  tbx_tick_t   t2;

  TBX_GET_TICK(t1);
  TBX_GET_TICK(t2);

  // initialise structs for datatype
  if (numberOfElements == 0) {
    for(i=0 ; i<blocks ; i++) {
      blocklengths[i] = 0;
    }
  }
  else {
    for(i=0 ; i<blocks ; i++) {
      blocklengths[i] = numberOfElements/blocks;
    }
    blocklengths[blocks-1] += numberOfElements % blocks;
  }

  displacements[0] = 0;
  for(i=1 ; i<blocks ; i++) {
    displacements[i] = blocklengths[i-1] + displacements[i-1];
  }

  if (use_hindex == TRUE) {
    for(i=0 ; i<blocks ; i++) {
      displacements[i] *= sizeof(float);
    }
  } // end if

  // create user datatype
  if (use_hindex == TRUE) {
    MPI_Type_hindexed(blocks, blocklengths, displacements, MPI_FLOAT, &indextype);
  }
  else {
    MPI_Type_indexed(blocks, blocklengths, displacements, MPI_FLOAT, &indextype);
  }

  MPI_Type_commit(&indextype);

  if (rank == source) {
    float        data[numberOfElements];
```

```
    // Initialise data to send
    if (VERBOSE) {
      printf("data = ");
    }
    for(i=0 ; i<numberOfElements ; i++) {
      data[i] = 1.0 * (i+1);
      if (VERBOSE) {
        printf("%3.1f ", data[i]);
      }
    }
    if (VERBOSE) {
      printf("\n");
    }

    // send the data to the processor 1
    TBX_GET_TICK(t1);
    MPI_Send(data, 1, indextype, dest, TAG, MPI_COMM_WORLD);

    // erase the local data
    for(i=0 ; i<numberOfElements ; i++) data[i] = -1.0;

    // receive data from processor 1
    MPI_Recv(data, numberOfElements, MPI_FLOAT, dest, TAG, MPI_COMM_WORLD, &stat);
    checkIndexIsCorrect(data, i, numberOfElements);

    TBX_GET_TICK(t2);
    fprintf(stderr, "%d\t%d\t%f\t%d\t%d-%d\n", numberOfElements, MPIR_INDEXED,
            TBX_TIMING_DELAY(t1, t2), blocks, source, dest);
  }
  else if (rank == dest) {
    float b[numberOfElements];
    MPI_Recv(b, numberOfElements, MPI_FLOAT, source, TAG, MPI_COMM_WORLD, &stat);
    checkIndexIsCorrect(b, rank, numberOfElements);

    MPI_Send(b, 1, indextype, source, TAG, MPI_COMM_WORLD);
  }

  MPI_Type_free(&indextype);
}
```

B.2 The Vector Datatype

```
int getRealSize(int size, int blocks) {
  int realSize = size;
  while (realSize % blocks != 0) realSize++;
  return realSize;
}

void sendVectorTypeFromSrcToDest(int size, int blocks, int rank, int source, int dest,
                                 int numtasks, int use_hvector) {
  int realSize = getRealSize(sqrt(size), blocks);
  float a[realSize][realSize];
```

```
MPI_Datatype columntype;
int i, j;
float *b;
MPI_Status stat;
tbx_tick_t    t1;
tbx_tick_t    t2;

int count = blocks;
int blocklength = realSize*realSize/blocks;
int stride = blocklength;

TBX_GET_TICK(t1);
TBX_GET_TICK(t2);

// Initialise data to send
for(i=0 ; i<realSize ; i++) {
  for(j=0 ; j<realSize ; j++) {
    a[i][j] = getValue(i, j, realSize);
    if (VERBOSE) {
      printf("%3.1f ", a[i][j]);
    }
  }
  if (VERBOSE) {
    printf("\n");
  }
}

// create user datatype
if (use_hvector == TRUE) {
  MPI_Type_hvector(count, blocklength, stride*sizeof(float), MPI_FLOAT, &columntype);
}
else {
  MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);
}
MPI_Type_commit(&columntype);

if (rank == source) {
  // send data to the process dest
  TBX_GET_TICK(t1);
  MPI_Send(&a[0][0], 1, columntype, dest, TAG, MPI_COMM_WORLD);

  // receive the data
  b = (float *) malloc(count*blocklength*sizeof(float));
  MPI_Recv(b, count*blocklength, MPI_FLOAT, dest, TAG, MPI_COMM_WORLD, &stat);
  checkVectorIsCorrect(b, rank, count, blocklength, size, stride);
  free(b);

  TBX_GET_TICK(t2);
  fprintf(stderr, "%d\t%d\t%f\t%d\t%d-%d\n", count*blocklength, MPIR_VECTOR,
          TBX_TIMING_DELAY(t1, t2), blocks, source, dest);
}
else if (rank == dest) {
  // receive the data
```

```
    b = (float *) malloc(count*blocklength*sizeof(float));
    MPI_Recv(b, count*blocklength, MPI_FLOAT, source, TAG, MPI_COMM_WORLD, &stat);

    checkVectorIsCorrect(b, rank, count, blocklength, realSize, stride);
    free(b);

    MPI_Send(&a[0][0], 1, columntype, source, TAG, MPI_COMM_WORLD);
  }

  MPI_Type_free(&columntype);
}
```

B.3 The Struct Datatype

```
typedef struct {
  float x, y;
  int c;
  float z;
} Particle;

void sendStructTypeFromSrcToDest(int numberOfElements, int rank, int source, int dest,
                                 int numtasks) {
  MPI_Datatype particletype;
  MPI_Aint offsets[3];
  MPI_Datatype oldtypes[3];
  int blockcounts[3];
  Particle *particles;
  Particle *p;
  int i;
  MPI_Status stat;

  tbx_tick_t   t1;
  tbx_tick_t   t2;

  TBX_GET_TICK(t1);
  TBX_GET_TICK(t2);

  // create the datatype to send and receive the data
  oldtypes[0] = MPI_FLOAT;
  oldtypes[1] = MPI_INT;
  oldtypes[2] = MPI_FLOAT;
  offsets[0] = 0;
  offsets[1] = 2 * sizeof(float);
  offsets[2] = offsets[1] + sizeof(int);
  blockcounts[0] = 2;
  blockcounts[1] = 1;
  blockcounts[2] = 1;

  /* Now define structured types and commit them */
  MPI_Type_struct(3, blockcounts, offsets, oldtypes, &particletype);
  MPI_Type_commit(&particletype);

  // Initialize the particle array and then send it to each task
```

```
  if (rank == source) {
    // Initialise data to send
    particles = (Particle *) malloc(numberOfElements * sizeof(Particle));
    for (i=0; i < numberOfElements; i++) {
      particles[i].x = (i+1) * 2.0;
      particles[i].y = (i+1) * -2.0;
      particles[i].c = (i+1) * 4;
      particles[i].z = (i+1) * 4.0;
    }

    TBX_GET_TICK(t1);
    MPI_Send(particles, numberOfElements, particletype, dest, TAG, MPI_COMM_WORLD);

    p = (Particle *) malloc(numberOfElements * sizeof(Particle));
    MPI_Recv(p, numberOfElements, particletype, dest, TAG, MPI_COMM_WORLD, &stat);
    //      checkStructIsCorrect(p, numberOfElements, rank);
    free(p);

    TBX_GET_TICK(t2);
    fprintf(stderr, "%d\t%d\t%f\t%d\t%d-%d\n", numberOfElements, MPIR_STRUCT,
            TBX_TIMING_DELAY(t1, t2), 3, source, dest);

    free(particles);
  } // end if
  else if (rank == dest) {
    p = (Particle *) malloc(numberOfElements * sizeof(Particle));
    MPI_Recv(p, numberOfElements, particletype, source, TAG, MPI_COMM_WORLD, &stat);
    checkStructIsCorrect(p, numberOfElements, rank);

    MPI_Send(p, numberOfElements, particletype, source, TAG, MPI_COMM_WORLD);

    free(p);
  } // end else
}
```

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

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 - 38334 Montbonnot Saint-Ismier (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 0249-0803