FLEX-MPI - Technical report

Gonzalo Martín[1], David E. Singh[1], Maria-Cristina Marinescu[2], and Jesús Carretero[1]

[1] Universidad Carlos III de Madrid, Spain
http://arcos.inf.uc3m.es
[2] Barcelona Supercomputing Center, Spain
http://www.bsc.es

1 Introduction

FLEX-MPI is a novel runtime approach for the dynamic load balancing of MPI-based SPMD iterative applications running on both homogeneous and heterogeneous platforms in the presence of dynamic external loads.

FLEX-MPI monitors the performance of the application and uses this information to make decisions with respect to the distribution of the workload and the data between processes, following a decentralized approach in which decisions are made collectively by all running processes. FLEX-MPI uses hardware counters and the MPI profiling interface to directly measure performance metrics at runtime.

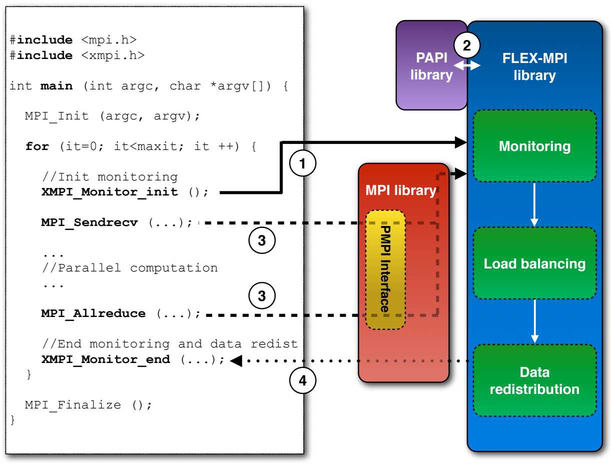
FLEX-MPI considers both burst and long-term external loads. Burst loads correspond to short-duration external loads which do not significatively affect the application performance. Long-term external loads reduce the application's CPU usage affecting its performance. FLEX-MPI is able to discriminate between these two kinds of loads and flexibly apply different load balance policies depending on the magnitude of the external load.

2 FLEX-MPI

FLEX-MPI is implemented as a library on top of the MPICH-2 [2] implementation. This makes it fully compatible with the MPI-2 features and allows it to easily link with any existing MPI-based application. Figure 1 illustrates the structure of a parallel application linked with the FLEX-MPI library, which integrates the functionalities for *monitoring* (arrows labeled 1, 2, and 3), *load balancing*, and *data redistribution* (arrow labeled 4).

2.1 Monitoring

The purpose of the monitoring functionality is to collect performance metrics for each process of the parallel application during its execution. FLEX-MPI collects performance metrics at runtime by means of hardware counters (via PAPI [3]), while it uses the MPI standard profiling interface (PMPI) to monitor the communication performance of the application.



```
#include <mpi.h>
#include <xmpi.h>

int main (int argc, char *argv[]) {

  MPI_Init (argc, argv);

  for (it=0; it<maxit; it ++) {

    //Init monitoring
    XMPI_Monitor_init ();

    MPI_Sendrecv (...);

    ...
    //Parallel computation
    ...

    MPI_Allreduce (...);

    //End monitoring and data redist
    XMPI_Monitor_end (...);
  }

  MPI_Finalize ();
}
```

Fig. 1. Structure and runtime calls of a parallel code linked with the FLEX-MPI library.

FLEX-MPI implementation uses low level PAPI interfaces (Table 2.1) to track the number of floating point operations $FLOP$, the real time $Treal$ (i.e. the wall-clock time), and the CPU time $Tcpu$ (i.e. the time during which the processor is running in user mode). FLEX-MPI assumes floating-point based applications which exhibit a correlation between the $FLOP$ and the workload size, which is reasonable for many parallel applications (e.g. linear system solvers).

Table 1. PAPI interfaces used by FLEX-MPI for monitoring.

PAPI function	Description
`PAPI_create_eventset (int *eventSet)`	Create an eventSet
`PAPI_add_event (int *eventSet,int eventCode)`	Add a hardware counter event (e.g. `PAPI_FP_OPS`)
`PAPI_remove_event (int *eventSet,int eventCode)`	Remove a hardware counter event
`PAPI_start (int *eventSet)`	Start the PAPI counting
`PAPI_stop (int *eventSet,long long *values)`	Stop the PAPI counting
`PAPI_read (int *eventSet,long long *values)`	Read a hardware counter
`PAPI_get_real_usec()`	Get real time
`PAPI_get_virt_usec()`	Get CPU time

2.2 Load balancing

The load balancing functionality computes the new distribution of the workload between the processes depending on the per-process values for the performance metrics measured via monitoring.

FLEX-MPI implements an algorithm which determines whether there exists external application load on each of the processing elements and the *relative computing power* of each process. Using this information, the load balancing algorithm computes the workload assigned to each process. Therefore, the algorithm leads to the most efficient data distribution depending on the actual performance of the processing elements.

2.3 Data redistribution

The data redistribution functionality is responsible for the mapping and transparent redistribution of the data between the processes when a load balance operation is carried out. This mechanism is necessary because in SPMD applications each process executes the same code but operates on a different subset of the data which is distributed without any replication among processes. However, FLEX-MPI also works for SPMD applications with replicated data. The data redistribution functionality considers both dense and sparse data structures, which need to be registered.

Section 3 describes the interfaces provided for registering data structures. Data redistribution operations use MPI standard messages and are optimized to perform data transferences in parallel between pairs of processes to minimize the transfer time and the redistribution overhead.

3 FLEX-MPI's Application Programming Interface

FLEX-MPI's API described in Table 3 permits to integrate the functionalities of the extension for monitoring, load balancing, and data redistribution in a parallel MPI-based application. We defined a set of standard `C` calls that need be added to a native MPI source code.

Table 2. FLEX-MPI's API.

FLEX-MPI's interface	Description
`XMPI_Get_wsize()`	Get the current workload size
`XMPI_Register_dense()`	Register a dense data structure
`XMPI_Register_sparse()`	Register a sparse data structure
`XMPI_Monitor_init()`	Start the monitoring functionality
`XMPI_Monitor_end()`	End the monitoring functionality

- `XMPI_Get_wsize(int rank, int size, int *count, int *displs, int **vcounts, int **displs)` function returns the current workload size of the process. This function requires the following parameters: (input) the rank of the process; (input) the number of running processes; (output) the current workload size; (output) the displacement of the workload within the data structure; (output) an array containing the size of the workload for each process; (output) an array containing the displacement of the workloads for each process.

- `XMPI_Register_dense(void *addr, int size)` registers a dense data structure. This function requires the following parameters: (input) the pointer to the data structure's address; (input) the size of the data structure (given as the number of elements).

- `XMPI_Register_sparse(void *addr_col, void *addr_row, void *addr_val, int size)` registers a sparse data structure. This function requires the following parameters: (input) the pointer to the array of columns index; (input) the pointer to the array of rows index; (input) the pointer to the array of values; (input) the size of the data structure (given as the number of rows).

- `XMPI_Monitor_init(void)` starts the monitoring functionality. This function does not require any parameter.

- `XMPI_Monitor_end(int rank, int size, int it, int *count, int *displs, int **vcounts, int **displs)` redistributes the workload according to the new workload distribution computed by the load balancing functionality. This function requires the following parameters: (input) the rank of the process; (input) the number of running processes; (input) the current iteration; (output) the new assigned workload size; (output) the displacement of the new workload within the data structure; (output) an array containing the size of the workload for each process; (output) an array containing the displacement of the workloads for each process.

4 FLEX-MPI installation

This section describes how to build, install, and work with FLEX-MPI. FLEX-MPI requires the `gcc` compiler, a previous installation of PAPI software, and MPICH-2. For our experiments, we worked with MPICH-2 version 1.4.1p1 under Linux Ubuntu 10.10.

The following commands are required to build and install FLEX-MPI:

1. Unpack the tar file:

```
tar xfz flex_mpi-v1.0.tar.gz
```

2. Build FLEX-MPI:

```
make all
```

3. Install FLEX-MPI in the current directory:

```
make install
```

Integrating FLEX-MPI in any MPI existing application requires to include the header file `xmpi.h` in the main file and linking it with the dynamic library `libxmpi.so` when compiling the program by adding the option `-lxmpi` (e.g. `mpicc -o jacobi jacobi.c -I../include/ -L../lib/ -lxmpi`).

5 Matrix analysis

In this section we present a detailed analysis of the matrices used for the performance evaluation tests for Jacobi and Conjugate Gradient.

For CG experiments we used matrices from the University of Florida sparse matrix collection [1]. Table 5 describes the characteristics of the matrices selected: *nd*24*k*, *ldoor*, and *audikw*_1. For each matrix, we show number of rows, density, and number of non-zero elements. We selected these matrices since this subset is representative of data structures which exhibit regular and irregular data distribution paterns.

Table 3. Description of sparse matrices used for CG.

Matrix	Size(rows)	Density(%)	NNZ
*nd*24*k*	72,000	0.005	28,715,634
ldoor	952,203	0.004	42,493,817
*audikw*_1	943,695	0.008	77,651,847

For Jacobi experiments we generated dense matrices with different sizes which characteristics are shown in Table 5. These matrices were synthetically generated with random values. Since they are dense matrices, all of the rows have the same exact number of non-zero entries, which leads to balanced data distributions.

The computation of CG depends on the number of non-zero entries assigned to each process, then using a static partitioning of an irregular sparse matrix leads to an unbalanced execution. This is because the basic distribution is a block distribution of the rows using blocks of the same length. Note that this

Table 4. Description of dense matrices used for Jacobi.

Size(rows)	NNZ
5,000	25,000,000
10,000	100,000,000
15,000	225,000,000

distribution leads to an unevenly distribution of the non-zero entries for matrices *nd*24*k* and *audikw*_1 (Figures 3 and 7). For instance, for an execution of CG on 4 processes with a block distribution based on the number of rows, the number of non-zero entries assigned to each process are shown in Table 5.

Table 5. NNZ distribution for a row-based block distribution for CG.

Matrix	P_0	P_1	P_2	P_3
*nd*24*k*	8,087,711	7,937,591	7,016,965	5,673,367
ldoor	10,742,828	11,362,221	10,578,883	9,809,885
*audikw*_1	28,820,601	15,279,927	16,743,579	16,807,740

5.1 Detailed analysis of sparse matrices used in CG

– $nd24k$ matrix exhibits an irregular data distribution pattern since the non-zero entries are unevenly distributed along the matrix. The number of non-zero entries decreases gradually depending on the row index. Figure 2 shows the data distribution pattern of the matrix, while Figure 3 shows the number of non-zero entries for each row.

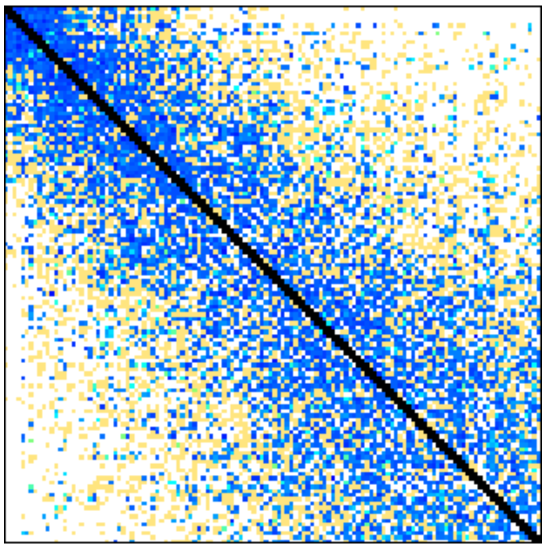


Fig. 2. $nd24k$ matrix.

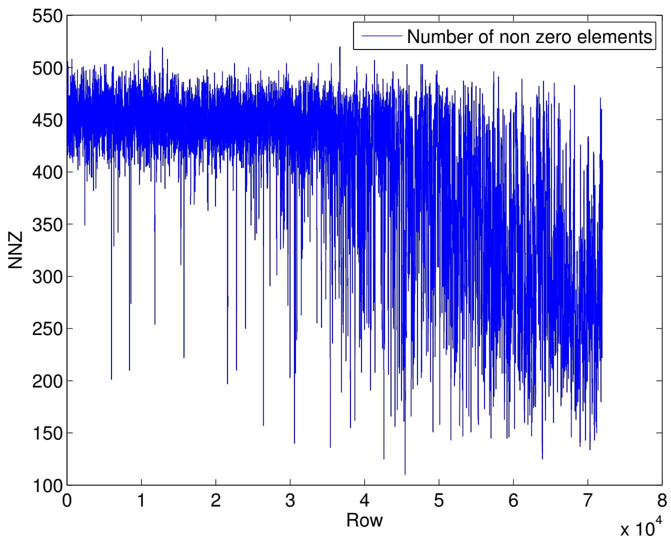


Fig. 3. Number or non-zero entries for each row of matrix $nd24k$.

– *ldoor* matrix exhibits a clustered data distribution pattern where the non-zero entries are evenly distributed along the matrix. Figure 4 shows the data distribution pattern of the matrix, while Figure 5 shows the number of non-zero entries for each row.

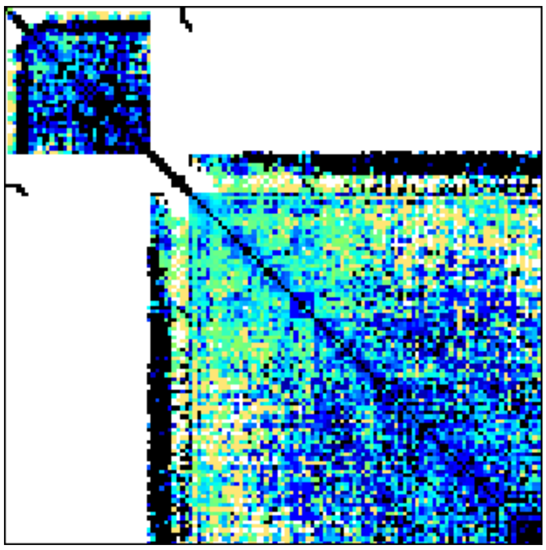


Fig. 4. *ldoor* matrix.

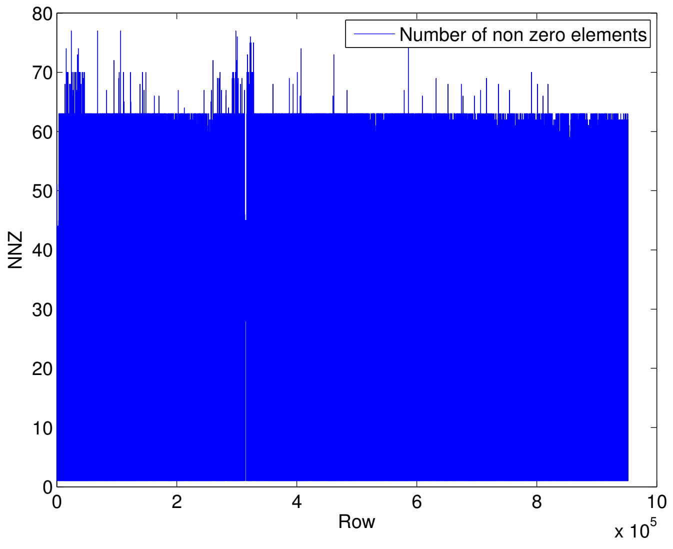


Fig. 5. Number or non-zero entries for each row of matrix *ldoor*.

– *audikw_1* matrix exhibits a partially banded data distribution pattern since the non-zero entries are unevenly distributed along the matrix. A large number of non-zero entries are distributed within the first rows of the matrix. Figure 6 shows the data distribution pattern of the matrix, while Figure 7 shows the number of non-zero entries for each row.

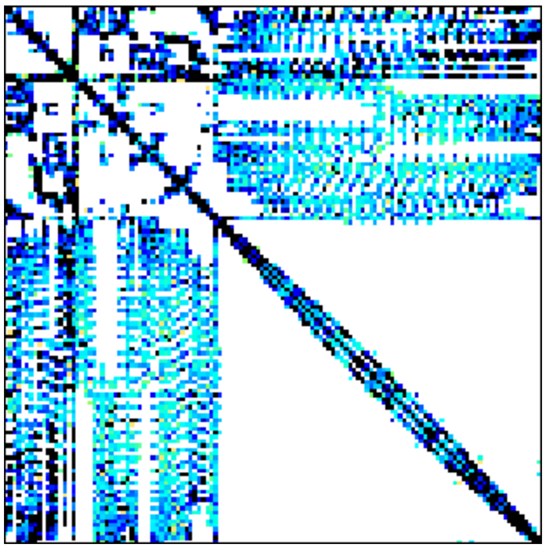


Fig. 6. *audikw_1* matrix.

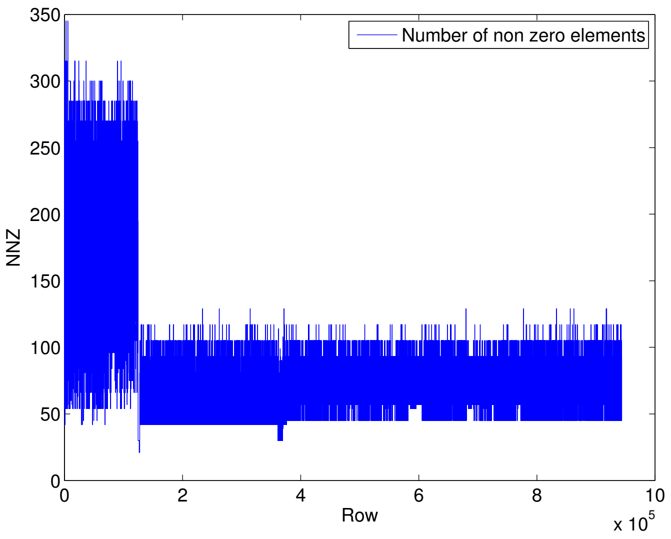


Fig. 7. Number or non-zero entries for each row of matrix *audikw_1*.

6 Code deployment

Only a few modifications are necessary to instrument MPI-based applications to integrate FLEX-MPI. This section shows the complete source codes of Jacobi and CG as examples of the instrumenting process of a MPI-based application.

Jacobi is an iterative method for solving a system of linear equations $Ax = b$ which operate on a symmetric dense matrix. The algorithm improves the solution in every iteration and stops when either the solution converges to a predefined tolerance or a predefined number of iterations is reached. Conjugate Gradient (CG) is an iterative linear system solver suited for large sparse matrices which must be symmetric and positive definite. The conditions under which a solution is reached are similar to Jacobi's.

Listings 1.1 and 1.3 show the source code for Jacobi and CG, while Listings 1.2 and 1.4 show the instrumented codes with FLEX-MPI interfaces (in bold). Note that, in both programs, only four function calls need to be added, in addition to the header file of the library.

6.1 Jacobi

```
#include <stdio.h>
#include <mpi.h>

...

//parallel jacobi
void jacobi (int dim, int rank, int size, double *A, double *b, double *x, double
     *x_old, int itmax, double diff_limit)
{
  double diff = 0, axi = 0;

  int n, m, it = 0, displ, count, tag=991, *displs = NULL, *vcounts = NULL;

  //memory allocation
  displs = (int*) malloc (*size * sizeof(int));
  vcounts = (int*) malloc (*size * sizeof(int));

  //get current worksize
  Get_wsize (rank, size, dim, &displ, &count, vcounts, displs);

  for (; it < itmax; it ++) {

    //matrix rows
    for (n = 0; n < count; n ++) {

      axi = 0;

      //matrix cols
      for (m = 0; m < dim; m ++) {

        axi += (A[(n*dim)+m] * x_old[m]);
      }

      x[n+desp] = x_old[n+desp] + (( b[n+desp] - axi ) / A[(n*dim)+n]);
    }

    for (diff = 0, n = desp; n < (desp+count); n ++)
      diff += fabs (x[n] - x_old[n]);

    //Allgather x vector
    MPI_Allgatherv (x+desp, count, MPI_DOUBLE, x_old, vcounts, displs, MPI_DOUBLE,
          MPI_COMM_WORLD);

    if (diff <= diff_limit) break;
  }

  if (rank == 0) printf ("[%i]␣Jacobi␣finished␣in␣%i␣iterations\n", rank, it);

  free (displs);  free (vcounts);
}
```

Listing 1.1. Jacobi native source code.

```c
#include <stdio.h>
#include <mpi.h>
#include <xmpi.h> //FLEX-MPI library

...

//parallel jacobi
void jacobi (int dim, int rank, int size, double *A, double *b, double *x, double
     *x_old, int itmax, double diff_limit)
{
  double diff = 0, axi = 0;

  int n, m, it = 0, displ, count, tag=991, *displs = NULL, *vcounts = NULL;

  //register data structures
  XMPI_Register_dense (A, dim*dim);
  XMPI_Register_dense (x, dim);

  //memory allocation
  displs = (int*) malloc (*size * sizeof(int));
  vcounts = (int*) malloc (*size * sizeof(int));

  //get current worksize
  XMPI_Get_wsize (rank, size, dim, &displ, &count, vcounts, displs);

  for (; it < itmax; it ++) {

    //monitor init
    XMPI_Monitor_init ();

    //matrix rows
    for (n = 0; n < count; n ++) {

      axi = 0;

      //matrix cols
      for (m = 0; m < dim; m ++) {

        axi += (A[(n*dim)+m] * x_old[m]);
      }

      x[n+desp] = x_old[n+desp] + (( b[n+desp] - axi ) / A[(n*dim)+n]);
    }

    for (diff = 0, n = desp; n < (desp+count); n ++)
      diff += fabs (x[n] - x_old[n]);

    //Allgather x vector
    MPI_Allgatherv (x+desp, count, MPI_DOUBLE, x_old, vcounts, displs, MPI_DOUBLE,
         MPI_COMM_WORLD);

    if (diff <= diff_limit) break;

    //monitor end
    XMPI_Monitor_end (rank, size, it, &count, &displ, &vcount, &displs);
  }

  if (rank == 0) printf ("[%i]␣Jacobi␣finished␣in␣%i␣iterations\n", rank, it);

  free (displs);
  free (vcounts);
}
```

Listing 1.2. Jacobi source code instrumented with FLEX-MPI.

6.2 Conjugate Gradient

```
#include <stdio.h>
#include <mpi.h>

//sparse matrix struct
typedef struct spr {
  int    dim;
  int    nnz;
  int    *row;
  int    *col;
  double *val;
} spr;

...

//parallel cg
void cg (int rank, int size, spr *A, double *x, double *b, int itmax, double
     diff_limit);
{
  double diff, rnorm2, rnorm2_old, rho, alpha, *s = NULL, *r = NULL, *z = NULL;

  int n, m, it = 0, displ, count, tag=991, *displs = NULL, *vcounts = NULL;

  //memory allocation
  displs = (int*) malloc (*size * sizeof(int));
  vcounts = (int*) malloc (*size * sizeof(int));
  s = (double*) malloc (A->dim * sizeof (double));
  r = (double*) malloc (A->dim * sizeof (double));
  z = (double*) malloc (A->dim * sizeof (double));

  //get current worksize
  Get_wsize (rank, size, dim, &displ, &count, vcounts, displs);

  for (; it < itmax; it ++) {

    if (it == 0) {

      memset (x, 0, (A->dim * sizeof (double)));

      //r = b - Ax
      matvec (s, A, x, desp, count, vcounts, displs);
      axpy (r, -1, s, b, A->dim, desp, count, vcounts, displs);

      memcpy (s, r, (A->dim * sizeof (double)));
      memset (z, 0, (A->dim * sizeof (double)));

      //r * r
      rnorm2_old = dot (r, r, desp, count);
    }

    //z = A * s
    matvec (z, A, s, desp, count, vcounts, displs);

    //alpha = rnorm2_old / s * z
    alpha = rnorm2_old / dot (s, z, desp, count);

    //x = alpha * s + x
    axpy (x, alpha, s, x, A->dim, desp, count, vcounts, displs);

    //r = -alpha * z + r
    axpy (r, -alpha, z, r, A->dim, desp, count, vcounts, displs);

    //r * r
    rnorm2 = dot (r, r, desp, count);

    //rho
    rho = rnorm2 / rnorm2_old;

    //s = rho * s + r
    axpy (s, rho, s, r, A->dim, desp, count, vcounts, displs);

    rnorm2_old = rnorm2;

    if (sqrt(rnorm2) <= diff_tol) break;
  }
  if (rank == 0) printf ("[%i]␣CG␣finished␣in␣%i␣iterations\n", rank, it);
  free (displs);  free (vcounts);  free (s);  free (r);  free (z);
}
```

Listing 1.3. Conjugate Gradient native source code.

```
#include <stdio.h>
#include <mpi.h>
#include <xmpi.h> //FLEX-MPI library

//sparse matrix struct
typedef struct spr {
  ...
} spr;

...

//parallel cg
void cg (int rank, int size, spr *A, double *x, double *b, int itmax, double
     diff_limit);
{
  double diff, rnorm2, rnorm2_old, rho, alpha, *s = NULL, *r = NULL, *z = NULL;

  int n, m, it = 0, displ, count, tag=991, *displs = NULL, *vcounts = NULL;

  //register data structures
  XMPI_Register_sparse (A->row, A->col, A->val, dim);
  XMPI_Register_dense (x, dim);

  //memory allocation
  displs = (int*) malloc (*size * sizeof(int));
  vcounts = (int*) malloc (*size * sizeof(int));
  s = (double*) malloc (A->dim * sizeof (double));
  r = (double*) malloc (A->dim * sizeof (double));
  z = (double*) malloc (A->dim * sizeof (double));

  //get current worksize
  XMPI_Get_wsize (rank, size, dim, &displ, &count, vcounts, displs);

  for (; it < itmax; it ++) {

    //monitor init
    XMPI_Monitor_init ();

    if (it == 0) {

      memset (x, 0, (A->dim * sizeof (double)));

      //r = b - Ax
      matvec (s, A, x, desp, count, vcounts, displs);
      axpy (r, -1, s, b, A->dim, desp, count, vcounts, displs);

      memcpy (s, r, (A->dim * sizeof (double)));
      memset (z, 0, (A->dim * sizeof (double)));

      //r * r
      rnorm2_old = dot (r, r, desp, count);
    }

    //z = A * s
    matvec (z, A, s, desp, count, vcounts, displs);

    //alpha = rnorm2_old / s * z
    alpha = rnorm2_old / dot (s, z, desp, count);

    //x = alpha * s + x
    axpy (x, alpha, s, x, A->dim, desp, count, vcounts, displs);

    //r = -alpha * z + r
    axpy (r, -alpha, z, r, A->dim, desp, count, vcounts, displs);

    //r * r
    rnorm2 = dot (r, r, desp, count);

    //rho
    rho = rnorm2 / rnorm2_old;

    //s = rho * s + r
    axpy (s, rho, s, r, A->dim, desp, count, vcounts, displs);

    rnorm2_old = rnorm2;

    if (sqrt(rnorm2) <= diff_tol) break;

    //monitor end
    XMPI_Monitor_end (rank, size, it, &count, &displ, &vcount, &displs);
  }
  if (rank == 0) printf ("[%i]␣CG␣finished␣in␣%i␣iterations\n", rank, it);
  free (displs);  free (vcounts);  free (s);  free (r);  free (z);
}
```

Listing 1.4. Conjugate Gradient source code instrumented with FLEX-MPI.

References

1. T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.
2. 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, 1996.
3. P. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.