Appendix D

Analysis and parallelization of the source code of EBNA$_{BIC}$

D.1 Short review of the source code of the sequential EDA program

Next, the parts of the code in the sequential EDA program that are more promising for parallelizing are described. This section is shown as an example of how parallelization has to be done and which are the main implications of it.

We will concentrate on the discrete case, more precisely in the EBNA algorithm and its way to compute the BIC score. As every node in the candidate Bayesian network does contribute to the overall BIC score of the structure, the score is computed first for each of the nodes. However, for our purpose we are not interested in the overall BIC score, but in finding Bayesian network that optimizes it. That is why we use a matrix `A`, where for every two nodes i and j of the Bayesian network, `A[i,j]` will record the change in the BIC score for the arc from i to j when it is added (if there is not an arc from i to j) or deleted (when there is that arc) in the Bayesian network. Other global variables in the sequential program that take part in the computation of the BIC score are the following:

IND_SIZE: the number of variables of the individuals. Therefore, this is also the number of nodes in the probabilistic graphical structure.

SEL_SIZE: the number of selected individuals from which the probabilistic graphical structure is to be learned.

cases: this matrix stores the selected individuals from which the learning is to be done. Its size is *SEL_SIZE* $\times$ *IND_SIZE*.

parents: this boolean matrix has a size of *IND_SIZE* $\times$ *IND_SIZE* and it is used to represent the Bayesian network: if i and j are two nodes, then when `parent[i,j]` is true it means j is the parent of i in the Bayesian network.

STATES: this integer vector of size IND_SIZE tell us the number of values that each variable can take.

The parts of the sequential program that perform the learning in EBNA are the following:

```
void CalculateA(int **&cases) {
  for(int i=0;i<IND_SIZE;i++)
        CalculateANode(i,cases);
}

void CalculateANode(int node, int **&cases) {
    double old_metric, new_metric;
    old_metric = BIC(node,cases);

    for(int i=0;i<IND_SIZE;i++)
      if (i!=node)
        {
            //change the j->i arc in the Bayesian network
            parents[node][i] = !parents[node][i];
            new_metric = BIC(node,cases);

            //restore the j->i arc in the Bayesian network
            parents[node][i] = !parents[node][i];

            //Compute difference
            A[node][i] = new_metric - old_metric;
        }
        else A[node][i] = INT_MIN;
}
```

And the actual BIC function is computed with the following function:

```
double BIC(int node, int **&cases) {
    int j,k;

    // Calculate the number of combinations of parent values.
    int no_j = 1;
    for(j=0;j<IND_SIZE;j++)
        if(parents[node][j]) no_j *= STATES[j];

    // Allocate memory for all nijk-s and initialize them.
    int ** nijk = new int*[no_j];
    for(j=0;j<no_j;j++)
    {
        nijk[j] = new int[STATES[node]];
        for(k=0;k<STATES[node];k++) nijk[j][k] = 0;
    }

    // Calculate all nijk-s.
    for(j=0;j<SEL_SIZE;j++)
    {
        // Find the parent configuration for the j-th case.
        int parent_configuration = 0;
        for(int parent=0;parent<IND_SIZE;parent++)
```

```
            if(m_parents[node][parent])
            {
                parent_configuration *= STATES[parent];
                parent_configuration += cases[j][parent];
            }

        // Update the corresponding nijk.
        nijk[parent_configuration][cases[j][node]]++;
    }

    // Calculate the BIC value.
    double bic = 0;
    for(j=0;j<no_j;j++)
    {
        int nij = 0;
        for(k=0;k<STATES[node];k++)
            nij += nijk[j][k];
    }

    bic -= log(SEL_SIZE)*no_j/2;

    // Free the memory allocated for the nijk-s.
    for(j=0;j<no_j;j++)
        delete [] nijk[j];
    delete [] nijk;

    return bic;
}
```

D.2 Parallelization using threads

The score+search procedure will be parallelized by making threads to divide the work: a thread plays the role of the *manager* that distributes the work among the rest, and the others are the *workers* that have to compute all the possible arc modifications for a same number of nodes (i.e each worker will execute the procedure `CalculateANode` for a total of `IND_SIZE/number_of_workers` nodes).

D.2.1 New adaptation on the source code for threads

The parallel program will use shared memory in the form of defined global variables for communication between the different cooperating threads. However, the use of shared variables for communication leads to the existence of race conditions within the program, and therefore a synchronization mechanism is required to ensure exclusive access to the critical sections in the program. The multithreading standard library selected is *pthreads*.

As already explained in Section 6.4.1, we apply a manager-slave working scheme. In our case, we decided to apply a manager-slave working scheme, where a thread plays the role of the manager and the rest of workers wait for a node number whose particular BIC score has to be computed. The worker threads will compute the BIC score by making use of common resources that are present in shared memory such as the global variables `cases`, `parents`, and `STATES`, and they will finally write their results in the shared matrix `A`.

In order to control the maximum number of threads that can be working at the same time a semaphore called `SemMaxChildren` is defined. This semaphore is initialized to the maximum number of threads that can exist. In our case we have a two processor computer, and therefore this limit was set to 4 threads.

In addition a table is created in order to store the thread identification for each worker-thread in the program. This one-dimensional table is called `WorkerId`, and it is used by the primitives to create and wait for threads.

Primitives to create and organize threads.

Having all this aspects in mind, some functions were defined in order to create an easier to use abstraction layer of the underlying low level pthreads interface. The calls to native *pthreads* primitives are easy to identify in the source code, as they all have the prefix `pthread_` on their names. These functions are intended to respect the maximum limit of threads established:

* **Creation of a thread.** This function allows a program to create a new thread within the process that executes it. If a call to the function `CreateThread` is done and there are already too many threads created, the function will block the call until one of the workers finishes, although this control is supposed to be done by the manager before trying to create a new thread. The last parameters of this function is the worker number of the thread, and the id of the newly created thread is stored in the table `WorkerId`. The function returns the number of the thread that is created, or -1 in case of error:

```
unsigned long CreateThread(void * (*my_func)(void *),
                           void *arglist, int NumWorker)
{
  pthread_t NumChildThread;
  pthread_attr_t thread_attr;
  int ThreadNr;
  void *(*functname)(void *) = (void *(*)( void * )) my_func;

  //making sure that there are not too many
  //threads already created
  sem_wait( &SemMaxChildren );
  num_threads++;

  status = pthread_attr_init (&thread_attr);
  if (status != 0)
      cerr << "Error " << status << ": Create attr" << endl;

  //Create a detached thread: this is required in order to
  //use more than a single CPU by a process
  status = pthread_attr_setdetachstate (&thread_attr,
                                        PTHREAD_CREATE_DETACHED);
```

```
    if (status != 0)
        printf("Error %d: Set detach", status);

    status = pthread_create (&NumChildThread, &thread_attr,
                             functname, arglist);
    if (status!=0) {
        printf("Error: the new thread could not be created!");
        return (-1);
    }

    //Set the worker number in the table
    WorkerId[NumWorker] = NumChildThread;

    return ((unsigned long) NumChildThread);
  }
```

* **End of a thread.** When a thread finishes its work it is destroyed by calling the function `EndThread`. However, if a thread is blocked waiting the end of another to be created, this function will unblock the call to `CreateThread`.

  As the *pthreads* library does not have any primitive to manage the end of the threads in a way that we need, we created a queue to store the end of threads within our application. Each time a thread finishes, the returned value is stored in the queue until a thread that is waiting for the end of another thread receives the information. This queue is a shared resource in memory, and therefore all the lines of code accessing it constitute a critical section. This critical section is managed with a mutex called `MutexQueue`. As this queue is a very typical example of a structure is memory that stores elements, the access to it has been represented in these two primitives: `insert_in_queue` and `read_from_queue`.

```
  void EndThread(void *value_ptr)
  {
    sem_post( &SemMaxChildren );

    //Record the end of the thread event to synchronize it with
    //the WaitingThread primitive
    wait( &MutexQueue);
    insert_in_queue(value_ptr, &EndedThreads);
    num_threads--;
    signal( &MutexQueue);

    sem_post( &SemWaitingAnyThread);

    pthread_exit(value_ptr);
    return;
  }
```

* **Kill another thread.** This function can be used to kill another thread. The id of the thread to kill must be provided. This function also stores a value in the queue to

control the end of processes, but in this case an error code of -1 is sent:

```
void KillThread(unsigned long ThreadNumber)
{
  pthread_kill( (pthread_t) ThreadNumber, SIGKILL);

  //Record the end of thread event
  wait( &MutexQueue);
  insert_in_queue(-1, &EndedThreads);//-1: code for errors
  num_threads--;
  signal( &MutexQueue);

  sem_post( &SemWaitingAnyThread);

  }

  return;
}
```

* **Waiting for a single thread.** This function is used to make threads wait for another's end. The only parameter required is the id of the thread that we are waiting for. If this parameters has the value 0, the function will wait for any of the existing threads. If no other thread is created within the process' environment, a error value of -1 is returned. It is also important to see that the tread will not be destroyed until another is ready to receive the end signal. This is controlled by synchronizing the ending thread with the someone waiting for a thread end by means of the semaphore `SemWaitingAnyThread` that is also used in the `EndThread` primitive described previously.

```
int WaitThread(unsigned long ThreadNumber)
{
  int ReturnValue;

  //If there are no threads left return an error
  if (num_threads==0) return(-1);


  sem_wait( &SemWaitingAnyThread);

  wait( &MutexQueue);
  if (ThreadNumber==0) {
      //The waiting thread is synchronized with the end
      //of someone else, and the returned value is given
      read_first_from_queue(&ReturnValue, &EndedThreads);
      return (int) ReturnValue;
  }
  else {
      signal( &MutexQueue );
      return thr_join(ThreadNumber, NULL, NULL);
      }
    }
```

* **Waiting for all threads.** This function is used to implement more easily the manager-slave scheme. It is used by the manager in order to keep waiting for the ending of all of the worker threads. This function has only a parameter which is the number of worker threads that were created, and it makes use of the table `Workerid` to obtain the id number of the next thread to wait for.

```
void WaitForAllThreads(int number_worker_threads)
{
  int i;

    //wait for all threads. If there is no left, return.
    if (m_num_threads==0) return;

    for ( i = 0; i < number_worker_threads; i++){
      pthread_join(m_Workerid[i], NULL);
      sem_wait (&SemWaitingAnyThread);
    }

    //This function does not use the queue, so empty it now
    wait( &MutexQueue);
    initialize_queue_to empty();
    signal( &MutexQueue);

    return;
}
```

The code of the manager and the workers.

Having defined all these primitives, we have now to change the sequential program and adapt it for the manager-slave working scheme. For this, two new functions are defined: `ParallelBIC` will be the one executing the master-thread, and `ParallelBICWorker` will be the one executing all the worker-threads. The function `ParallelBIC` will have two arguments: which are the number of nodes that the Bayesian network has (this is also the number of tasks to do in parallel) and the number of the node that we are computing each time. In fact, the function in the sequential program where the job is divided is `CalculateANode` in which all the relationships with the rest of the nodes are computed using the BIC score, and the function `ParallelBIC` will divide the whole task in pieces giving each worker the corresponding amount of work. We will use some global variables for communication between the manager and the slaves (`CurrentTask`, `MaxNumTasks`, `Node`, `OldMetric` and `JobSize`). In addition, a mutex called `MutexComm` is used for ensuring the mutual exclusion when accessing critical variables used for communication.

```
void ParallelBIC(int NumTasks, int node)
//NumTasks is the total amount of nodes
//Node is the node number that we are processing in parallel.
```

```
{
   //Initialize global variables
   CurrentTask =0; //we start from the node 0
   Node = node; //number of the node that we are treating
   JobSize = ((NumTasks-1) / MAX_THREADS)+1;
   MaxNumTasks = NumTasks;

   //The actual value of the metric is computed
   //so that the workers compare it with their
   //work. This is a sequential task that
   //cannot be parallelized.
   OldMetric = BIC(Node, cases);

   // Creation of the parallel crew by using threads
   for(int i=0;i<NumTasks;i++)
   {
       WorkerNum=num_threads;
       CreateThread((void * (*)(void *)) ParallelWorkerBIC, NULL);
   }

   //Wait until all the last threads have finished their work.
   WaitForAllThreads(MAX_THREADS);

   return;
}

void ParallelWorkerBIC(void) {
   int FirstJob, LastJob;
   double new_metric;

   //Calculate the part of the work to do - critical section
   wait( &MutexQueue);
   FirstJob = (CurrentTask) * (JobSize);
   LastJob = ((CurrentTask + 1) * (JobSize)) -1;
   CurrentTask++;
   signal( &MutexQueue);

   //Check that the end of the work is not reached
   if (LastJob > MaxNumTasks) LastJob = MaxNumTasks;

   for(int i=FirstJob;i<LastJob;i++) {
     if (i!=Node) {

         //a new function is used to compute the difference of
         //changing the arc i-> node. Here we cannot change
         //the variable parents as this is shared with the rest
         // of the threads
```

```
        new_metric = deltaBIC(Node,i, cases);

        A[Node][i] = new_metric - OldMetric;
      }
      else A[Node][i] = INT_MIN;
    }

    EndThread(NULL);
    return;
}
```

The new function deltaBIC that is used by the workers to compute the difference in the BIC score if we modify the arc that in the nodes *nparent-¿node* by adding it (if it does not exist in the Bayesian network) or removing it (if it does exist in the Bayesian network). Thereof the name of the function, which is very similar to the BIC function, essentially it does only change the consideration of the nparent value on the computation. However, here we compute the difference in the partial BIC score of the node node without changing the value of the global variable parents. This last aspect is very important as the variable parents is shared among all the working-threads and any change on it will also corrupt the computation for the rest of the working-threads:

```
double deltaBIC(int node, int nparent, int **&cases) {int j,k;

   // Calculate the number of parent configurations.
   int no_j = 1;
   for(j=0;j<IND_SIZE;j++) {
     //If we are analyzing nparent, consider it as changed
     if (j!=nparent) {
       if(parents[node][j]) no_j *= STATES[j];
     }
     else {
       if(!(m_parents[node][nparent])) no_j *= STATES[nparent];
     }
   }

   // Allocate memory for all nijk-s.
   int ** nijk = new int*[no_j];
   for(j=0;j<no_j;j++)
   {
       nijk[j] = new int[STATES[node]];
       for(k=0;k<STATES[node];k++) nijk[j][k] = 0;
   }

   // Calculate all nijk-s.
   for(j=0;j<SEL_SIZE;j++)
   {
       // Find the parent configuration for the j-th case.
       int parent_configuration = 0;
       for(int parent=0;parent<IND_SIZE;parent++) {
         //If we are analyzing nparent, consider it as changed
         if (parent!=nparent) {
           if(m_parents[node][parent])
           {
               parent_configuration *= STATES[parent];
               parent_configuration += cases[j][parent];
           }
         }
         else {
           if(!(m_parents[node][nparent]))
           {
               parent_configuration *= STATES[nparent];
               parent_configuration += cases[j][nparent];
           }
         }
       }

       // Update the corresponding nijk.
       nijk[parent_configuration][cases[j][node]]++;
   }

   // Calculate the BIC value.
   double deltabic = 0;
   for(j=0;j<no_j;j++)
   {
       int nij = 0;
       for(k=0;k<STATES[node];k++) nij += nijk[j][k];
   }

   deltabic -= log(SEL_SIZE)*no_j/2;

   // Free the memory allocated for the nijk-s.
   for(j=0;j<no_j;j++)
       delete [] nijk[j];
   delete [] nijk;

   return deltabic;
}
```

We also have to initilialize all the structures and shared variables at the beginning of the program in the following way:

```
#include <limits.h> #include <semaphore.h> #include <pthread.h>

#define MAX_THREADS  4  /* Max Number of Worker-threads
                           since we have 2 processors */

... pthread_t thread; int num_threads=0; //this variable is
updated automatically
```

```
                   //in CreateThread and EndThread
int JobSize, CurrentTask, MaxNumTasks, Node; double OldMetric;
          //Variables for communication between master and workers

pthread_mutex_t MutexQueue; pthread_mutex_t MutexComm; sem_t
SemMaxChildren, SemWaitingAnyThread; int WorkerId[MAX_THREADS];
... main() {
  ...
  //Initialise semaphores and mutex
  sem_init(&SemMaxChildren, 0, MAX_THREADS);
  pthread_mutex_init(&MutexQueue, NULL) ;
  pthread_mutex_init(&MutexComm, NULL) ;
  sem_init(&SemWaitingAnyThread, 0, 0);
  ...
}
```

And finally, we have to change the `CalculateANode` function of the sequential version of the program so that it executes the parallel version instead of the sequential one. This time it requires only one parameter, as the `cases` matrix has been converted as a global variable in order to be accessed by all the threads at the same time:

```
void CalculateANode(int node) {
    ParallelBIC(IND_SIZE, node);
}
```

This example shows how to adapt a sequential program to convert it as parallel. We have used the BIC score as an example, but any other scores for the discreet case could also use the same idea and they would easily be adapted following these steps. Furthermore, the continuous version of the EDA program can also be adapted in a similar way. The continuous EDAs EGNA$_{BIC}$ and EGNA$_{BGe}$ where also parallelized following a similar approach.

D.3 Parallelization using MPI

MPI has been designed to use message passing as the communication mechanism for inter-process communication. MPI provides an efficient mechanism for threads from different processes, and it can also been applied even when shared memory is available.

As we have seen in the previous section, the parallel version of EBNA using threads makes use of shared variables not only for communication, but also for reading the data required for the `ParallelWorker` function. The fact of using shared memory also required the use of external synchronization mechanisms such as semaphores.

On the other hand, in the particular example of the EDA program, the fact that processes cannot share any memory among them requires the manager to send all the data structures required to compute the `deltaBIC` function to each of the workers. The data structures that have to be sent by the manager to the workers every time we create a worker-process are the matrices `cases` and `parents`, the vector `STATES`, and the global variables `IND_SIZE` and `SEL_SIZE`. These variables have a length according to the size of the problem, and when applying EDAs for a big example they can increase in size rapidly as well as the amount of information to communicate to the workers.

D.3.1 New adaptation on the source code for MPI

The main function of the source code is organized as follows: MPI creates all the processes at the beginning of the execution, so it is important to make the workers wait and to make the master start the initialization phase and to make it execute all the sequential parts of the program. We have designed a communication protocol for communicating the master and the workers that contains two steps: firstly, the manager will send an integer to all the workers in order to inform them about the next operation they have to perform, where 0 means to compute the BIC score in parallel and 1 means to finish their execution and terminate. If we wanted the workers to parallelize another second part of the EDA program we could also have created a new operation and added another work-code to the protocol. If the program is to be finished, workers will end after receiving the 0 value and then the manager will prepare the final results. If the BIC score is to be computed in parallel, the master will have to send to all the workers all the data matrix, vectors and variables they require for performing their job, and then the manager will receive all the results from all the workers in order to gather all the parts of the `A` matrix computed in parallel by the workers.

The amount of work that each worker has to do will depend on the number of workers and the size of the matrix `A`, and therefore the division on the amount of work for each process will be computed automatically taking these aspects into account. The main function of the source code that carries out this approach is the following:

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

  //Initialize the MPI system
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &g_size); //g_size <- number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //my_rank <- process MPI id

  NumWorkers = g_size; //1 manager-worker, n-1 workers

  if (my_rank == 0) {
    //Code for the Master

    //The initilization CommInitializeBICParallel_MPI() is done once
    //the parameters are read on the program

    EDA(argc, argv); // The manager reads the arguments and executes
                     // the part of the program reserved for the
                     // master.

    //At the end of the program the master send the code 0 to
    //indicate the workers that they have finished their work.
    op = 0;
    MPI_Bcast(&op, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
  else {
    //Code for the Workers
    //Initializes the variables required for the rest of the work.
    CommInitializeBICParallel_MPI() ;

    while (1){ //Execute once and again...

      //wait until receiving a code of operation to perform
      MPI_Bcast(&op, 1, MPI_INT, 0, MPI_COMM_WORLD);

      switch(op) {  //depending on the operation-code

      case 1: //Compute the BIC score
              CommStartBICParallel_MPI(); //read all the data
                                          //structures required
              //Calculate which is the corresponding part of A
              //for this worker to compute
              for (i=0; i<WorkSize; i++)
                   ParallelWorkerBIC(my_rank*WorkSize+i, i);

              //Send the results
              CommEndBICParallel_MPI();

              break;
      case 0: //exit the program
      default: break;
      }

      if (op==0) break; //end the while.
    }
  }

  if (my_rank == 0) {

    //Write final results
    WriteFinalResults();
  }

  MPI_Finalize();
}
```

The initialization function at the beginning of the program is executed by all the processes, either the manager and the workers, and it is written as follows:

```
void CommInitializeBICParallel_MPI() { int i,j;

  //The manager computes the size of work for each worker
  if (my_rank ==0) {
    WorkSize = IND_SIZE/NumWorkers;
  }
  //Receive IND_SIZE
  MPI_Bcast(&IND_SIZE, 1, MPI_INT, 0, MPI_COMM_WORLD);
  //Receive SEL_SIZE
  MPI_Bcast(&SEL_SIZE, 1, MPI_INT, 0, MPI_COMM_WORLD);
  //Receive Worksize
  MPI_Bcast(&WorkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  //Reserve memory for global variables
  parents_MPI = (bool**) malloc (IND_SIZE*sizeof(bool *));
  A_MPI = (double **) malloc (IND_SIZE*sizeof(double *));
  for (i=0; i<IND_SIZE; i++) {
    parents_MPI[i] = (bool *) malloc (IND_SIZE*sizeof(bool));
    A_MPI[i] = (double *) malloc (IND_SIZE*sizeof(double));
  }
  cases_MPI = (int **) malloc (SEL_SIZE*sizeof(int *));
  for (i=0; i<SEL_SIZE; i++) {
    cases_MPI[i] = (int *) malloc (IND_SIZE*sizeof(int));
  }
  STATES_MPI =  (int *) malloc (IND_SIZE*sizeof(int));

  //Receive STATES
  if (my_rank ==0) { // The manager prepares the matrix STATES to be sent
    for (i=0; i<IND_SIZE; i++)
      STATES_MPI[i] = STATES[i];
  }
  MPI_Bcast(STATES_MPI, IND_SIZE, MPI_INT, 0, MPI_COMM_WORLD);

  return;
}
```

The function that will be executed at the beginning of the parallel BIC operation will send all the required input data that changes from a generation to the next to all the workers. The function executed at the end of the parallel BIC operation will send all the results to the manager. These two functions are implemented as follows:

```
void CommStartBICParallel_MPI() { int i,j;

 //Receive parents
 MPI_Bcast(parents_MPI, IND_SIZE*IND_SIZE, MPI_INT, 0, MPI_COMM_WORLD);

 //Receive cases
 MPI_Bcast(cases_MPI, SEL_SIZE*IND_SIZE, MPI_INT, 0, MPI_COMM_WORLD);

}

void CommEndBICParallel_MPI() { int i,j;

 //Gather the Results
 MPI_Gather(A_MPI, WorkSize*IND_SIZE, MPI_DOUBLE, A_MPI,
            WorkSize*IND_SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);

}
```

Note that these functions are short in source code, but in fact require a high overload of exchanged information, as some of the matrixes that are sent such as the `cases` can have a very big size. Therefore, the time to send them to the workers can be of considerable importance. It is also important to realize that these functions where not required when using threads, as these were shared in memory among all the workers.

All the functions seen up to now where just to exchange data between the master and the slaves. All of them were not required in the parallel version of the BIC program with threads. The following functions are the ones executed by the master and the workers respectively in order to perform the parallel BIC task. The main difference on this new version with MPI is that the master will also play the role of a worker at the same time, and this is done in such a way because the creation of a new process and sending all the input variables to it is very time consuming:

```
void MPI_ParallelBIC(int NumTasks)
//The Manager
{
  int i,j;

  //The manager organizes the work:
  //Send the signal for job to all the workers
  op = 1;
  MPI_Bcast(&op, 1, MPI_INT, 0, MPI_COMM_WORLD);

  CommStartBICParallel_MPI();

  //The master also takes part on the computation
  //of a part of the parallel BIC similarly as
  //the rest of the workers
  for (i=0; i<WorkSize; i++)
       ParallelWorkerBIC(my_rank*WorkSize+i, i);

  //Gather Results from the rest of workers
  CommEndBICParallel_MPI();

  //Now all the results are stored in the matrix A
}


void ParallelWorkerBIC(int node, int TaskNumber) {
    double old_metric, new_metric;
    old_metric = BIC(node);

    for(int i=0;i<IND_SIZE;i++)
```

```
        if(i!=node) {
           new_metric = deltaBIC(node,i);
           A_MPI[TaskNumber][i] = new_metric - old_metric;
        }
        else A_MPI[TaskNumber][i] = INT_MIN;

    return;
}
```

where the `BIC` and `deltaBIC` functions are defined exactly as in the parallel case with threads.

Finally, just a short mention of the inclusion of the following lines on the program

```
#include <limits.h>
#include "mpi.h"
```