בְּ"ה
PARA2000, Bergen, Norway, June 2000

Experiments in Separating Computational Algorithm from Program Distribution and Communication

R.B. Yehezkael[1] (Formerly Haskell) , Y. Wiseman[1,2] ,
H.G. Mendelbaum[1,3], I.L. Gordin[1]

[1] Jerusalem College of Technology, Computer Eng. Dept., POB 16031, Jerusalem 91160, Israel
E-mail:rafi@mail.jct.ac.il, Fax: 009722-6751-200
[2] University Bar-Ilan, Math. and Computer Sc. Dept., Ramat-Gan 52900, Israel
[3] Univ. Paris V, Institut Universitaire de Technologie, 143-av. Versailles, Paris 75016, France

Abstract
Our proposal has the following key features:
1)   The separation of a distributed program into a pure algorithm (PurAl) and a distribution/communication declaration (DUAL). This yields flexible programs capable of handling different kinds of data/program distribution with no change to the pure algorithm.
2)   Implicit or automatic handling of communication via externally mapped variables and generalizations of assignment and reference to these variables. This provides unified device independent view and processing of internal data and external distributed data at the user programming language level.
3)   Programs need only know of the direct binds with distributed correspondents (mailbox driver, file manager, remote task, window manager etc.). This avoids the need for a central description of all the interconnections.
The main short-range benefits of this proposal are to facilitate parallel computations. Parallel programming is a fundamental challenge in computer science, nowadays. Improving these techniques will lead to simplify the programming, eliminate the communication statements, and unify the various communication by using an implicit method for the transfer of data which is becoming essential with the proliferation of distributed networked environment. We present 2 experiments of separation between PurAl and DUAL, using a preprocessor or an object-type library. This new approach might be of interest to both academic and industrial researchers.

1 INTRODUCTION

In many cases the same algorithm can take various forms depending on the location of the data and the distribution of the code. The programmer is obliged to take into account the configuration of the distributed system and modify the algorithm in consequence in order to introduce explicit communication requests. In a previous paper[21], we proposed to develop implicit communication and program distribution on the network, here we present two experiments in this field.

There are two problems: distribution of data, and distribution of code. Either a piece of code can be copied in various nodes and work with distributed data, or the whole code is split in various nodes and each part works with distributed data. Similiarly, data can be located locally and copied in various sites, or split and distributed over various places. Let us concentrate in the more common case of components of code located in different nodes and transfering data to each other.

With the proliferation and increased use of networks, it has become very important to develop software which can easily access information in a distributed networked environment. Our aim is to facilitate this easy access to data at the programming language level.

Aim:
We want to use any "pure" algorithm written as if its data were in a local virtual memory, and run it either with local or remote data without changing the source code.
1)   This will simplify the programming stage, abstracting the concept of data location and access mode.
2)   This will eliminate the communication statements since the algorithm is written as if the data were in local virtual memory.
3)   This will unify the various kinds of communication by using a single implicit method for transfer of data in various execution contexts: concurrently executing programs in the same computer, or programs on several network nodes, or between a program and remote/local file manager, etc.

For example, the programmer would write something like: x := y + 1;
where y can be "read" from a local true memory (simple variable), or from a file, or from an edit-field of a Man-Machine-Interface (for example, WINDOWS/DIALOG BOX), or through communication with another parallel task, or through a network communication with another memory in another computer. In the same way, x can be "assigned" a value (simple local variable) or "recorded" in a file, or "written" in a window, or "sent" to another computer, or "pipe-lined" to another process in the same computer.

2 PROPOSAL

We propose to separate a program into a pure algorithm and a declarative description of the links with the external data.
1)   the "pure algorithm" (PurAl) would be written in any procedural language without using I/O statements.
2)   A "Distribution, Use, Access, and Linking" declarative description (DUAL declaration) is used to describe the way the external data are linked to the variables of the pure algorithm. (i.e. some of the variables of the pure algorithm are externally mapped.) The DUAL declaration is also used to describe the distribution of the programs.

To summarize in the spirit of Wirth [20] :
    Program = Pure Algorithm + DUAL declaration of distribution and External Data access
or even briefer:
    Program = PurAl + DUAL

2.1 The Producer Consumer Example and Variations

Pure Algorithm 1:
```
#include "external_types.h"
void   c ( )
{vec seq1;                    // vec is defined in external_types.h
 for (int i=1;  ; i++)
        {try {    data_processing (seq1 [i]);           // A
             }
         catch (out_of_range_error  ce) {break;};   // when up-bound exception on seq1,  i.e. end of data
         }
}   //end c
```

Informal example of Separate DUAL declarations for c : declarations of Distribution, Use, Access, and Linking, (see two kinds of different concrete implementation in section 5)
c'site is on computer1;
c.seq1'site is on mailbox2;
c.seq1'access is of type IN with sequential_increasing_subscript;
c.seq1'locking policy is gradual;

Pure Algorithm 2:

```
#include "external_types.h"
void   p ( )
{vec seq2;                    // vec is defined in external_types.h
 for (int j=1;  ; j++)
         { if (exit_condition) break;
           seq2 [j]: = <expression>; // B
         }
} // end p
```

Informal example of Separate DUAL declarations for p : declarations of Distribution, Use, Access, and Linking, (see two kinds of different concrete implementation in section 5)
p'site is on computer2;
p.seq2'site is on mailbox2;
p.seq2'access is of type OUT with sequential_increasing_subscript;
p.seq2'locking policy is gradual;

Some Comments on the above Programs

The first two algorithms are written at the user programmer level and we have hidden the details of the communication in the package "external_types.h". Local declarations are used inside a loop to define which elements of a vector, are being processed. The variables seq1 and seq2 are externally mapped by the DUAL declarations. The type vec is for a vector of characters of undefined length: this means that externally the vector is unconstrained, but internally the bounds, which are given by the local current value of "i or j", define which elements are being processed. The exception mechanism is used to detect the "end of data" condition ("out_of_range_error").
        In the line marked "// A " of the function c, seq1[i] is referenced and this causes a value to be fetched from outside, according to the DUAL declaration where c.seq1'site indicates the location of the vector, a mailbox. Also c.seq1'access indicates the way of accessing the data, "IN sequential_increasing_subscript" means that seq1 is read from outside in sequential manner using a sequentially increasing subscript. Similarly in the line marked "// B " of the function p, the vector seq2[j] is assigned and this causes a value to be sent outside the program according to the DUAL declaration where p.seq2'site indicates the location of the vector, a mailbox. Also regarding p.seq2'access, "OUT sequential_increasing_subscript" means that seq2 is written outside in sequential manner using a sequentially increasing subscript.

With the previous DUAL declarations, for seq1 and seq2, the programs will behave as a producer/consumer when run concurrently.
        Other distributed use of the same algorithms are possible by only modifying the DUAL declarations. With the modifications below, they are run as completely independent procedures, with for example seq1 coming from a file and seq2 being put onto a screen. The function c and p are not changed at all.

Modified execution using another DUAL declaration for c
c.seq1'site is on local_computer.file3;
c.seq1'access is of type IN with sequential_increasing_subscript;
c.seq1'locking policy is gradual;
c'site is on comp1;

Modified execution using another DUAL declaration for p
p.seq2'site is on local_computer.screen;
p.seq2'access is of type OUT with sequential_increasing_subscript;
p.seq2'locking policy is gradual;
p'site is on comp2;

3 HANDLING DATA DISTRIBUTION : UNIVERSAL VIRTUAL MEMORY (UVM)

The realization of our proposal makes the distributed data available to various local/remote programs. The data looks like they were in a unique pseudo-local-memory which is mapped to a Universal Virtual Memory (UVM). The UVM is located at various distributed physical addresses. So, if the data are really located in the local physical memory, the access is immediate. If they are located on disk, it can be viewed as a persistent storage. If they are sent or received to or from another device or remote memory, through a port, it can be viewed as communication etc.

Addressing:

In figure 1, one can see several distributed programs exchanging data on a network, each one using a software layer of UVM support based on mapping tables giving a general UVM addressing space.
Based on the DUAL declaration, the local virtual address is mapped to a UVM public name which consists of a UVM site name and a local name. In short, an externally mapped variable is associated with a UVM public name, where:
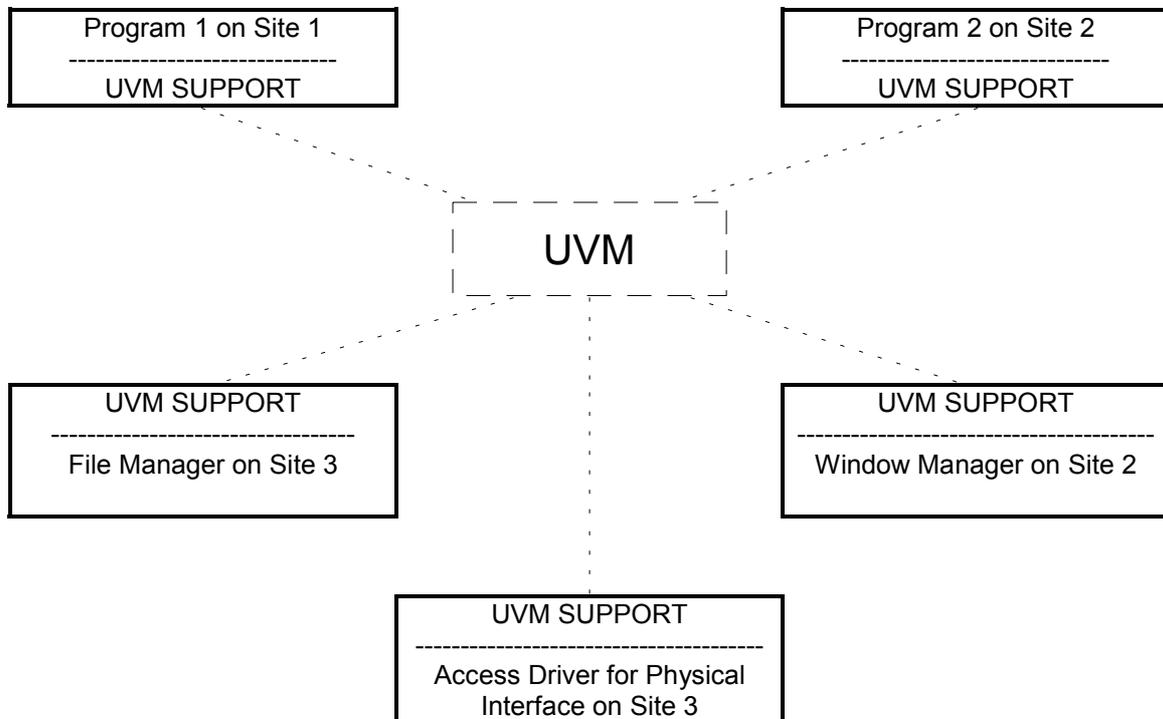UVM public name = UVM site + local name.

FIGURE 1: LOGICAL VIEW OF PROGRAMS AND SYSTEM COMPONENTS INTERACTING WITH EACH OTHER VIA UVM SUPPORT

4 HANDLING CODE DISTRIBUTION

Typically when a program is initiated, it will use the DUAL declaration of the mailboxes it uses to send a request to activate associated (secondary local or remote) programs and bind their externally mapped variables. These associated (secondary) programs would likewise activate associated (tertiary) programs and bind their externally mapped variables, and so on. At initialization time, a copy of (relevant parts of) the DUAL declaration are sent to associated programs for compatibility checking (types, format, access mode, locking strategies) and mapping table building. Algorithmic processing does not start until all interconnections are established and compatibility checks completed.

4.1 Using Already Existing Distribution

Let us consider a network with already existing general purpose programs on various sites: for instance a file manager, an algorithm for FFT computation, etc. We can then use the DUAL declarations to connect them so that the exchange of data can be properly coordinated and synchronized (like in our producer/consumer examples, see 2.1).

If the application is not already distributed on the network, we can have two ways of building the parallelism for the execution, statically at initialization time and dynamically at runtime.

4.2 Static Distribution at Initialization Time

Suppose we have a specific application in which there are several programs which need to be distributed on a network. The distribution can be handled from one site which will use the DUAL declarations to send the various programs to various sites and build the connections between them.

4.3 Dynamic Distribution at Run TIme

In some applications we do not know in advance the exact location of the procedures which need to be copied and distributed. In this case, the DUAL can indicate the location of the main procedure and data at initialization time, but at run time the procedure calls will cause the automatic distribution of procedures on available processors.

5 EXAMPLES OF IMPLEMENTATION PROTOTYPES

The DUAL syntax need not to be exactly as it was shown in the section 2.1, it depends on the system where it is realized. To implement the idea of separating the algorithm description and the distribution/communication declarations, we tried two ways : building a C- preprocessor for Unix, or building a C^{++} object-type library.

5.1 C-Unix-Preprocessor experiment

In this case the DUAL declarations will be sentences put at the beginning of the program or at the beginning of a bloc. For this method we wrote a preprocessor that reads the DUAL distribution/communication declarations and adds the synchronization, distribution and I/O statements to the pure algorithm (PurAl) according to these DUAL declarations. It fits to static declaration of data and program distribution before execution.

Let's take the classical merge_sort algorithm, the PurAl is written as if the sources and the target were in a virtual local memory without I/O and without explicit distribution calls. Using DUAL declarations, we define separately the two sources and the target as external_types. Furthermore, using a DUAL declaration, we can also define the parallelism of some functions separately from the algorithm, so that the algorithm is written purely as if it was sequential.

Please see below this specific DUAL syntax in this case :

Pure Algorithm 3 : Merge_sort with its DUAL declarations:

```
DUAL parallel      merging   in  one_level; // declaration for the parallelization of the
#define N  10       // algorithmic function 'merging' located in the function 'one_level'
void  main ( )
{ DUAL devin    source_data "./data";  // declaration : the 'source_data' variable is external
                                  // and is linked to the input sequential file "./data"
  DUAL devout  target "/dev/tty";      // declaration : the 'target' variable is external
                                  // and is linked to the output screen "/dev/tty "
  int      i;  char  x[N ];            // local memory vector  x   to perform the sort
        strncpy ( x , source_data , N  );  // the source_data file is copied to the local memory x
        msort ( x , N  );                  // performing the sort on the local memory vector x
        strncpy ( target , x , N  );      // the sorted vector x is copied to the target screen
} //end main------------------------------------------------------------
void  msort (char *x, int n )          // the merge sort algorithm will sort x using split subvectors aux
{ DUAL share    aux;      // declaration : all the split subvectors aux will share the same memory
  int      i, size=1;          // initial size of the subvectors
        while ( size < n )
        {       one_level ( x, aux, n , size ); // prepares the subvectors merges at this level of size
                strncpy ( x , aux , n  );      // copy back the sorted subvectors aux to vector x
                size*=2;                      // growing the size  of the subvectors
        }
}//end msort-----------------------------------------------------
```

```
void one_level ( char *x, char *aux, int n , int size )    //prepares the subvectors to merge
{// 1st vector :     lb1, ub1=low and up bounds;        2nd vector :       lb2, ub2=low and up bounds;
  int     lb1=0, ub1, lb2, ub2, lb3=0, i;                // lb3 = low bound of the 3rd (merged) vector
        while ( lb1+size < n  )
             {      lb2=lb1+size;
                    ub1=lb2-1;
                    if ( ub1+size >= n  )        ub2=n -1;
                    else                          ub2=ub1+size;
                    merging ( x, aux, lb1, lb2, lb3, ub1, ub2 );// can be done sequentially or in parallel
                    lb1=ub2+1;
                    lb3+=size*2;
             }
        i=lb1;   while ( lb3 < n )  aux[lb3++]=x[i++];    //The rest of the vector is copied
}//end one_level-----------------------------------------------------------
void merging ( char *x, char *aux, int lb1, int lb2, int lb3, int ub1, int ub2 )  //performs the real merge
{int I=lb1, j=lb2, k=lb3;                                  // of one subvector
        while (( i <= ub1 ) && ( j <= ub2 ))
                    if ( x[i] < x[j] )    aux[k++]=x[i++];
                    else               aux[k++]=x[j++];
        while ( i <= ub1 )    aux[k++]=x[i++];
        while ( j <= ub2 )    aux[k++]=x[j++];
}// end merging
```

Explanations :

The above text of the Pure Algorithm 3 is preprocessed before C-compiling. The preprocessor detects the DUAL declarations and inserts, in the text of the algorithm, the explicit I/O, parallelization and synchronization statements, corresponding to a Unix environment.

a) DUAL for implicit communication (implicit references to I/O variables) :

For instance, when the preprocessor finds : DUAL devin source_data "./data";

it will replace it with the following UNIX compatible text

char * source_data=(_devinp[_devin_i]=fopen ("./data" , "r"), fgets ((char *) malloc (BUFSIZ) , BUFSIZ , _devinp[_devin_i++]));

In the same manner, when the preprocessor finds :   DUAL devout target "/dev/tty"; ";

it will replace it with the following UNIX compatible text

char * s=(_devoutp[_devout_i++]=fopen ("/dev/tty" , "w"), (char *) malloc (BUFSIZ)) ;

And when source_data or target are used in the algorithm, the preprocessor will insert the necessary I/O statements, for instance at the end of the main, it will add the UNIX compatible

text :   fputs (target , _devoutp[-- _devout_i]); free (target); fclose (_devoutp[_devout_i]);
        free (source_data); fclose (_devinp[-- _devin_i]);

b) DUAL for implicit use of shared local memory :

For instance, when the preprocessor finds :   DUAL share   aux;

it will replace it with the following UNIX compatible text

char * aux=(char *)shmat (_shmid[_share_i++]=shmget(IPC_PRIVATE , BUFSIZ ,0600) , 0, 0600) ;

And at the end of the same bloc, the preprocessor will add

shmdt (aux); shmctl (_shmid[-- _share_i] , IPC_RMID , 0);

c) DUAL for implicit use of parallelism and synchronization :

For instance, when the preprocessor finds the text   DUAL parallel merging in one_level;

it will add at the beginning of the one_level function, the following text

int parallels=parallel[++parallelI]=0;   //to prepare the necessary number of parallel branches

Then, the preprocessor will add the parallelization UNIX statement 'fork' before calling the merging functions

_parallel[_parallel_i]++; if (fork( ) == 0) { merging(x, aux, lb1, lb2, lb3, ub1, ub2); exit (0); }

Finally, at the end of the one_level function, the preprocessor will add the synchronization UNIX statement wait according to the number of parallel branches

for (;_parallel[_parallel_i]>0;_parallel[_parallel_i]--) wait(&_parallel_s); _parallel_i--;

The appendix I gives the full C-program generated after preprocessing this Pure Algorithm 3 .

5.2 C⁺⁺ Object-type library experiment

In this case the DUAL declarations will be variables declarations (using predefined "external_types") put at the beginning of the program or at the beginning of a block. For this method, we wrote a C⁺⁺ Object-type library which handles the distributed variables, implements and masks all the devices declarations and I/O statements, using the C⁺⁺ possibility of overloading the operators. It fits well to dynamic external variable linking, but not to program distribution. Ravid [22] has also made such an experiment in her thesis. Please see below this specific DUAL syntax in this case :

<u>Pure algorithm 4 of sorting elements by the increment index method</u>.

```
#include "External_types.h"          // C⁺⁺ Object-type library (see below appendix II)
//DUAL declarations to handle the variables linked to implicit communications :
        DUALext_float a(in_out,file,"aa",ran);
        DUALext_float b(in_out,file,"bb",ran);
        DUALext_float t(in_out,file,"tt",ran);
        DUALext_float keyboard(in,console);
        DUALext_float screenFloat(out,console);
        DUALext_char screenChar(out,console);
void main( )
        { int n , i, j;                        // local variables
        screenChar =" Enter number of elements:\n ";        // sends message to the 'screen'
        n = keyboard;            // reads the number of elements to sort from the 'keyboard'
                                        // and puts it in the local variable 'n '
        screenChar =" Enter your elements:\n ";            // sends message to the 'screen'
        for( i=0;i<n ;i++) a[i]=keyboard[i];// reads from the 'keyboard', the elements to sort
                                        // and stores them in the file a
        for( i=0;i<n ;i++)                            // algorithm
                for( j =0;j < n ;j++)          // sorts the elements a[i] using the indexes vector t[i]
                        if (a[i]>a[j]) t[i]=t[i]+1;
        for (i=0;i<n ;i++) b[t[i]]=a[i];        // stores the sorted elements in the file 'b'
        for (i=0;i<n ;i++) {screenChar='\n '; screenFloat[i]=b[i];} //prints sorted file 'b' on 'screen'
        }
```

Explanations :

We have written a package " External_types.h " which contains a set of class-types "DUALext_float" "DUALext_int" "DUALext_char" "DUALext_double" etc… which permits to declare variables that can be linked to various devices with various access modes : when you declare an external variable, in a Pure Algorithm, you invoke the constructor of the corresponding class and indicate through the parameters the type of link you desire, the device, the access mode etc… In the above text of the Pure Algorithm 4, there are some DUAL declarations. For instance,

DUALext_float a (in_out,file,"aa",ran);   declares a variable 'a' of type float, which is linked to a random (direct-access) file named "aa" which can be read or written (in_out).

DUALext_float keyboard (in,console);   declares a variable ' keyboard ' of type float, which is linked to the input device of the console.

DUALext_float screen (out,console);   declares a variable ' screen ' of type float, which is linked to the output device of the console.

<u>Object-type library</u>

In the Appendix II, we give the full class DUALext_float , one can see the enum declarations and the constructor functions which permits to declare the deviceName if the variable has to be linked with a console, a file, or a network port (through com1 or com2). One can declare if he wants an input link (using the parameter in), an output link (using the parameter out), or both (using in_out). If the link is with a file, one has to declare the access mode using the parameter ran for random direct access files, or seq for sequential access.

When the external variable is used in the Pure Algorithm, it invokes automatically one of the overloaded operators '=' to output automatically values to the desired device, or the overloaded '[ ] ' for indexing I/O values, or the casting operator to provoke automatic input of values from the desired device.

So in this type of implementation, there is no special compilation, the C⁺⁺ compiler translates as usual. The declarations and the implicit communications are done at run-time.

The other classes DUALext_int, DUALext_char, DUALext_long etc.. are built on the same principle.

6 RELATED WORKS AND DISCUSSION

6.1 Transparent Communication

Kramer et al. proposed to introduce interface languages (CONIC [14], REX [15], DARWIN [16], Magee and Dulay [17]) which allows the user to describe centrally and in a declarative form the distribution of the processes and data links on a network. But the algorithm of each process contain explicit communication primitives.

Hayes et al [18] working on MLP (Mixed Language Programming) proposes using remote procedure calls (RPC's) by export/import of procedure names.

Purtilo [19] proposed a software bus system (Polylith) also allowing independence between configuration (which he calls "Application structure") and algorithms (which he calls "individual components"). The specification of how components or modules communicate is claimed to be independent of the component writing, but the program uses explicit calls to functions that can be remote (RPC) or local.

The Darwin, MLP, and Polylith are oriented towards a centralized description of an application distributed on a dedicated network. So all the binds and instances of programs are defined initially at configuration time.

Our approach in DUAL is aimed towards a non dedicated network in which each program knows only of the direct binds with its direct correspondents. Our claim is that our approach is better suited to interconnected programs in a non dedicated network.

6.2 Implicit File Handling

In the early 1980's the PDP/11 BASIC implementation included a virtual array feature, which identifies an array with a disk file. With this approach, files could be handled transparently (i.e. no read/write statements but instead assignment and reference to array variables). This corresponds to our approach although it was limited to disk files only, the man machine interface, sequential access to data, and communication not being treated at that time. In the same way, some operating systems can access files via virtual memory mapping e.g. VAX/VMS and some versions of UNIX.

Similarly in persistent storage systems (e.g. the E programming language [13]), we find that assignment and reference are used as a "file" access method for persistent data. The possibility of using these statements to handle communication is not discussed in [13].

Functional and Logic programming languages[3,4,12] use streams for mapping sequential data files onto lists. Indeed, in the LUCID[5] language, every variable is a stream or sequence of values. This too is a partial approach, treating only external lists and sequences via memory mapping. Our approach is more general in that we provide a unified notation for sequential and random access modes to arrays.

6.3 Handling the Man Machine Interface Transparently

Separating the man machine interface from the programming language has been extensively discussed over the years[2] (Hurley and Sibert 1989)[11]. Some researchers considered the application part as the controlling component and the user interface functions as the slave. Others do the opposite: the user interface is viewed as the master and calls the application when needed by the I/O process. Some works (Parnas 1969)[8] described the user interface by means of state diagrams. Edmonds (1992)[9] reports that some researchers describe the user interface by means of a grammar. Some others presented an extension of existing languages, Lafuente and Gries (1989)[10].

CONCLUSION

in our approach, the user interface and the application are defined separately and the link between them is explicitly but separately described. This approach is more general in that the user interface is seen as one part of a unified mechanism in which external data are accessed, the other parts of this unified mechanism being file handling, I/O, and network communication. This separation of the DUAL distribution declaration makes the Pure Algorithm (PurAl) clearer, independent of a network configuration, and versatile in the sense that it can be run in various contexts. The 2 experiments we conduct, show that this idea is feasible and can help the easier development of distributed applications.

Appendix I : executable merge_sort for Unix :
full C-program generated after preprocessing the Pure Algorithm 3
(in italic are the additions of the preprocessor to translate the DUALs)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/msg.h>
#include <string.h>
#include <malloc.h>
#define MAXEXVAR 100
int _share_i , _shmid[MAXEXVAR]; int _filed_i , _fd[MAXEXVAR];
struct stat _buf[1]; int _send_i , _smsqid[MAXEXVAR];  struct msgbuf _msgp[1];
char *_msgtext;  int _rcv_i , _rmsqid[MAXEXVAR];  int _devout_i;
FILE *_devoutp[MAXEXVAR]; int _devin_i; FILE *_devinp[MAXEXVAR];
int _parallel_i=-1 , _parallel[MAXEXVAR];


void main( )        // translation of DUAL source_data and target
{char * source_data =( _devinp[_devin_i]=fopen ( "./data" , "r" ) , fgets ( ( char * )
         malloc ( BUFSIZ ) , BUFSIZ , _devinp[_devin_i++] ) );
  char * target =( _devoutp[_devout_i++]=fopen ( "/dev/tty" , "w" ) , ( char * )
        malloc ( BUFSIZ ) ) ;
  int      i;   char  x[10 ];
        strncpy ( x , source_data , 10  );
        msort ( x , 10  );
        strncpy ( target  , x , 10  );
fputs ( target  , _devoutp[-- _devout_i] );
free ( target); fclose (_devoutp[_devout_i] ); free(source_data );fclose (_devinp[--_devin_i] );
}// end main---------------------
void msort(char *x,int n )          //translation of DUAL share
{ char * aux=(char *)shmat (_shmid[_share_i++]=shmget ( IPC_PRIVATE, BUFSIZ, 0600 ) ,
        0 , 0600 ) ;
        int      i,size=1;
        while ( size < n  )
        {      one_level ( x, aux, n , size );
               strncpy ( x , aux , n  );
               size*=2;
        }
shmdt ( aux );  shmctl ( _shmid[-- _share_i] , IPC_RMID , 0 );  //end translation DUAL share
}// end msort---------------------------------------------------------
void one_level ( char *x, char *aux, int n , int size )
{ int _parallel_s=_parallel[++_parallel_i]=0;      //begin translation DUAL parallel
        int      lb1,lb2,lb3,ub1,ub2,i;
        int      stat;
        lb1=lb3=0;
        while ( lb1+size < n  )
        {      lb2=lb1+size;  ub1=lb2-1;
               if ( ub1+size >= n  ) ub2=n -1;
               else            ub2=ub1+size;
               _parallel[_parallel_i]++; // translation DUAL parallel  merging
               if ( fork() == 0 ) {
                       merging( x, aux, lb1, lb2, lb3, ub1, ub2);
                       exit (0);}
               lb1=ub2+1;
               lb3+=size*2;
        }
        i=lb1;  while ( lb3 < n  )  aux[lb3++]=x[i++];  //end translation DUAL parallel
for (;_parallel[_parallel_i]>0;_parallel[_parallel_i]--) wait(&_parallel_s);  _parallel_i--;
}// end one_level------------------------------------------------
```

```
       void merging ( char *x, char *aux, int lb1, int lb2, int lb3, int ub1, int ub2 )
       {int i, j, k;        i=lb1;   j=lb2;   k=lb3;
               while (( i <= ub1 ) && ( j <= ub2 ))
                               if ( x[i] < x[j] )    aux[k++]=x[i++];
                               else                  aux[k++]=x[j++];
               while ( i <= ub1 )   aux[k++]=x[i++];
               while ( j <= ub2 )   aux[k++]=x[j++];
       }// end merging----------------------------------------
```

Appendix II : part of the external_types.h package for PC :

```
#include "bys1.h"           // enables the link of the variables with the serial ports COM1 and COM2
#include <fstream.h>        // enables the link of the variables with files
#include <iostream.h>
#include <conio.h>          // enables the link of the variables with with standard I/Os of the computer
#include <stdio.h>
#include <stdlib.h>
enum inout{in,out,in_out}; enum deviceName {console,file,com1,com2 };enum AccessMode{seq,ran};
int       init_com1 = 0 ,
class DUALext_float                      //--------EXTERNAL   F L O A T   TYPE----------
{inout IO; deviceName  DEVICE; char* FileN; AccessMode AccessM; float Value; long index;
 public: DUALext_float(inout, deviceName , char*, AccessMode);     //constructors
       DUALext_float(inout,deviceName );
       ~DUALext_float( );                         // destructor
       void operator=(float);              // overloading of operators for implicit communication
       DUALext_float  operator[ ](long);
       operator float( );
}; /////////////////////////////////////////////////////////////////            //CONSTRUCTOR  for files
DUALext_float::DUALext_float(inout D, deviceName  A, char* FileName, AccessMode C)
{IO = D; DEVICE = A; FileN = FileName; AccessM = C; Value = 0;index=0; }
DUALext_float:: DUALext_float(inout D, deviceName  A)
{IO = D; DEVICE  = A;               //CONSTRUCTOR  for console,com1,com2
       if (DEVICE  == com1)
       { if (!init_com1) open_net( );
         init_com1++;
       }
}//----------------------------------------------------------
DUALext _float::~ DUALext _float( )                          // DESTRUCTOR
{       if (DEVICE  == com1){ init_com1--;  if (!init_com1) close_net( ); }
}//------------------------------------------------------------//OVERLOADING OF '[ ]'; for INDEXING
DUALext_float   DUALext ext_float::operator[ ](long count ){
{       if (DEVICE  == file) count *=14;
       if ( AccessM == seq && count  < index){cerr <<"Error in sequential .\n "; exit(1); }
       index = count ;
       return(*this);
}//------------------------------------------------------------//OVERLOADING OF '=' ; OUTPUT OF VALUE
void DUALext   DUALext _float::operator=(float value)
{ if (IO != in){    if ( DEVICE == console) { cout << value;}
               if ( DEVICE  == file){    ofstream to; to.open(Ch,ios::ate); to.seekp(index);
                                         to.width(13); to << value <<' '; to.close( ); }
               if ( DEVICE  == com1) send_net(value);
           }
  return;
}
```

```
DUALext_float::operator float( )              //OVERLOADING OF CASTING : INPUT OF VALUE
{       if (DEVICE  == console) { cin >> Value; }
        if (DEVICE  == file ) {ifstream from;  from.open(Ch,ios::ate); from.seekg(index);
                                from.width(13); from >> Value; from.close(); }
        if (DEVICE  == com1 && IO == in)
                        { if ( something_in_net( ))Value = receive_net( ); else Value = 0;}
        return(Value);
}//----------------------------------------------------------------
```

BIBLIOGRAPHY

[1] "Hermes: A Language for Distributed Computing", Robert E. Strom et.al. Research Report, IBM T.J. Watson Research Center, Oct. 18th, 1990.

[2] "The Separable User Interface", Editor Ernest Edmonds, Academic Press, 1992.

[3] "Report on the Programming Language Haskell, A Non-strict Purely Functional Language", Paul Hudak et.al., Yale University Research Report No. YALEU/DCS/RR-777, 1st March 1992.

[4] "Concurrent Prolog - Collected Papers Vols 1,2" edited by Ehud Shapiro, MIT Press, 1987.

[5] "LUCID, the Dataflow Programming Language", W.W. Wadge and E.A. Ashcroft, Academic Press, 1988.

[6] "Res Edit Complete", P. Alley and C. Strange, Addison Wesley, 1991.

[7] "Interface Builder", Expertelligence Corp., 1987.

[8] "On the Use of Transition Diagrams in the design of a User Interface for an Interactive Computer System", D.L. Parnas, Proceedings of the National ACM Conference 1969, pp. 379-385, also appears in [2].

[9] "Emergence of the Separable User Interface", E. Edmonds, appears as the introduction of the book he edited, see [2].

[10] "Language Facilities for Programming User-Computer Dialogues", J.M. Lafuente and D. Gries, IBM Journal of Research and Development 1978, Vol. 22 No. 2, pp. 122-125, also appears in [2].

[11] "Modelling User Interface-Application Interactions", W.D. Hurley and J.L. Sibert, IEEE Software, January 1989, pp. 71-77, also appears in [2].

[12] "Functional Programming: Application and Implementation", P. Henderson, Prentice Hall, 1980"

[13] "The Design of the E Programming Language", J.E. Richardson, M.J. Carey and D.T. Schuh, Research Report, Computer Sciences Department, University of Winconsin.

[14] "Constructing Distributed Systems in Conic", J. Magee, J. Kramer, and M.S. Sloman, IEEE Transactions on Software Engineering, Vol 15 No. 6, pp 663 - 675, 1989.

[15] "An Introduction to Distributed Programming in REX", J. Kramer, J. Magee, M. Sloman, N . Dulay, S.C. Cheung, S. Crane, and K. Twiddle, in "Proceedings of Esprit, Brussels, 1991.

[16] "Structuring Parallel and Distributed Programs", J. Magee, N . Dulay, and J. Kramer, in "Proceedings of the International Workshop on Configurable Distributed Systems, London, 1992.

[17] "MP: A Programming Environment for Multicomputers", J. Magee and N . Dulay, appears in "Programming Environments for Parallel Computers", edited by N . Topham, R. Ibbett, and T. Bemmerl, Elsevier Science Publishers B.V. (North Holland), 1992

[18] "A Simple System for Constructing Distributed Mixed Language Programs", R. Hayes, S.W. Manweiler, and R.D. Schlichting, Software Practice and Experience, Vol 18, No. 7, pp 641 - 660, 1988.

[19] "The Polylith Software Bus", J.M. Purtilo, ACM Transactions on Programming Languages and Systems, Vol 16 No. 1, pp 151 - 174, 1994

[20] "Algorithms + Data Structures = Programs", N . Wirth, Prentice Hall, 1976

[21] "Toward Implicit Communication and Program Distribution" H.G. Mendelbaum, R.B. Yehezkael (Haskell) IEEE workshop on Computer Sc., Univ. Bar-Ilan, 1996

[22] "External Variables as a Unifying Concept for Handling Input/Output and Interprogram Communication", P. Ravid, Master's degree in the Dept. of Computer Sc., Univ. Bar-Ilan, 1995