Exploiting Single-Assignment Properties to Optimize Message-Passing Programs by Code Transformations

Alfredo Cristóbal-Salas[1], Andrey Chernykh[2], Edelmira Rodríguez-Alcantar[3], and Jean-Luc Gaudiot[4] (*)

[1] School of Chemistry Science and Engineering, Autonomous University of Baja California, Tijuana, Baja California, Mexico, 22390
cristobal@uabc.mx

[2] Computer Science Department, CICESE Research Center, Ensenada, Baja California, Mexico, 22830
chernykh@cicese.mx

[3] Computer Science; University of Sonora,Hermosillo, Sonora, Mexico, 83000
edelmira@mat.uson.mx

[4] Electrical Engineering and Computer Science, University of California, Irvine, Irvine, California, USA, 92697
gaudiot@uci.edu

Abstract. The message-passing paradigm is now widely accepted and used mainly for inter-process communication in distributed memory parallel systems. However, one of its disadvantages is the high cost associated with the data exchange. Therefore, in this paper, we describe a message-passing optimization technique based on the exploitation of single-assignment and constant information properties to reduce the number of communications. Similar to the more general partial evaluation approach, technique evaluates local and remote memory operations when only part of the input is known or available; it further specializes the program with respect to the input data. It is applied to the programs, which use a distributed single-assignment memory system. Experimental results show a considerable speedup in programs running in computer systems with slow interconnection networks. We also show that single assignment memory systems can have better network latency tolerance and the overhead introduced by its management can be hidden.

1 Introduction

The exchange of information remains as a critical bottleneck in distributed memory systems. Exchanging information by message passing is a popular technique in distributed environment. Furthermore, with the proliferation of clusters and GRID technology, the message passing paradigm has significantly increased in popularity. However, its major drawback is the inherently high communication costs. Communication cost depends on memory manipulation overhead (message preparation, message interpretation) and network communication delays.

(*) Authors are listed in alphabetical order.

There are several strategies to minimize this cost such as computation and communication overlapping, network optimization, or reduction of number of messages (message coalescing, caching messages, etc). Consequently, reducing this cost is vital to achieve good performance.

In this paper we present how to reduce communication cost of parallel programs for distributed memory systems. Technique eliminates synchronization issues by non-strict data access and fully asynchronous operations. It also combines functional programming techniques such: I-Structures [2] and partial evaluation [11] together with classical program optimization like constant-propagation, loop unrolling and dead-code elimination. As a contribution of this paper, we provide detailed description about code transformations needed to partially evaluate memory accesses when part of the program's input information is available. We use single-assignment I-Structures to facilitate asynchronous access when structure production and consumption can be allowed to proceed with a looser synchronization. When a read operation occurs before a write operation, the deferred request is queued on a linked list of that particular I-Structure element. When the write operation finally occurs, the system responds to the deferred reads by distributing the written value to the requesters, which have been received in the meantime.

On the other hand, partial evaluation [11,18] is an automatic program transformation technique which allows the partial execution of a program when only some of its input data are available (static), and specializes it by pre-computing parts of the program that depend on specific parameter settings. It has been shown in [9, 14] that the majority of communications in scientific programs are static, that is, the communication information can be determined at compile time. Some experiments which show how MPI parallel programs can be optimized by using static information can be found in [20]. These characteristics can be exploited in message passing paradigm to eliminate memory request at compile time. Elimination of memory accesses may improve performance of parallel programs running in architectures with high latency interconnection networks such as wide area networks or grids. Even though our technique works directly with MPI as communication layer, it can be applied to other communication libraries.

The rest of the paper is organized as follows: in section 2 a general description of proposed optimization technique is presented. In section 3, we provide detailed information how optimization technique works using an example of code transformation. Experimental results can be found in section 4. Related work is presented in section 5. Finally, some conclusions are presented.

2 General Description of the Optimization Technique

In [8], this optimization technique is proposed. This technique is based on a particular case of partial evaluation approach where parallel programs evaluation is performed when only part of their input is given. It reduces the number of messages in single-assignment distributed memory systems by exploiting constant information. For instance, matrix multiplication can be evaluated when matrices size and number of processes are known, but with unknown matrices elements values. Obviously, program evaluation cannot be completed but it is possible to create a residual program (optimized one). When remaining input data become available, residual program can continue evaluations. This residual program can be run as many times as needed, and it is expected to be faster than executing the original program.

Fig. 1 shows a general view of this new technique. Parallel program code and a set of constant values are given as an input. The output is a residual (optimized) code where all constant memory accesses have been eliminated. Two main steps are considered: *pre-processing* and *message elimination*.

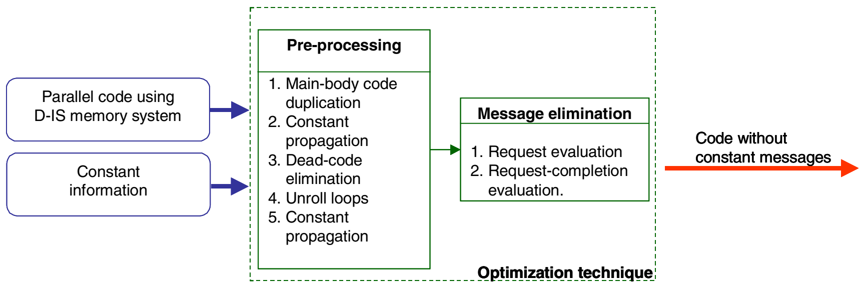


Fig. 1. General view of the optimization technique

In the *preprocessing* step, code is transformed to facilitate detection of static memory accesses. Main-body code is replicated in accordance with the number of processes given, constants are propagated, dead code is eliminated, and loops are unrolled.

In the *message elimination* step, static memory accesses are evaluated by inserting a special instruction in the corresponding remote process code to locally perform the remote request. After the evaluation of all static memory requests, a second review of code is performed to complete execution of all requests that refer to elements already defined. Before going into details, we review design of Distributed I-Structure memory system. More information about it can be found in [6, 7].

2.1 Distributed I-Structure Memory System (D-IS)

D-IS is a communication library for distributed memory systems that implements the functionality of I-Structures [2] on top of MPI (Fig. 2). Each MPI process manages a local I-Structure memory system arranged in a linked list. Remote operations are performed using split-phase transactions and they are implemented using MPI point-to-point routine calls. Exchange of information involves a *send-request*, *receive-value* on the requester side and *receive-request*, and *send-value* on the side of the owner of the I-Structure. D-IS permits consulting an I-Structure element even before a value is bound to that memory location. This feature breaks the restrictions unnecessarily imposed by sequential systems, which demand the complete production of data before consumption. The write policy is write-through to ensure data will be available as soon it is produced. D-IS is a further research of the I-Structure memory system presented in [15]. As D-IS runs on top of MPI, it has most of its features such as portability and efficient implementation in several architectures. The D-IS memory system has been tested in a NUMA S2MP ORIGIN 2000 and in a Pentium III cluster.

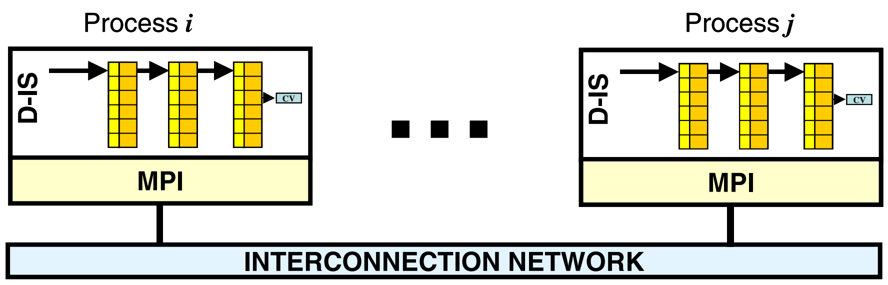


Fig. 2. Graphical representation of the D-IS

3 Functionality of Optimization Technique

Before presenting functionality of proposed technique, we first describe the syntax of the main function routines.

3.1 Syntax of Instructions to Manipulate D-IS Memory System

D-IS has four general routines to initialize memory system and to obtain general information from the communicator:

- `void DIS_Init(int argc, char **argv)`. Initializes the D-IS memory system. `argc` and `argv` are parameters taken from the command line.
- `void DIS_GetProcessRank(int *rank)`. Gets the rank of a process inside the current communicator.
- `void DIS_Finalize()`. Finalizes the D-IS memory system and stops the execution of all MPI routines.

The D-IS memory system also has the following instructions

- `int DIS_Request(int node, int id, int pos)`. It requests the element `pos` of the I-Structure `id` to process `node`. Remote requests are stored in a list whose index is attached to a MPI message as a continuation vector. This routine returns the position of the request in the list.
- `void DIS_RecvRequest(int node)`. This instruction is divided into three steps. First, an *MPI_Recv* instruction is executed to receive a request. Secondly, local D-IS is consulted to obtain information about the I-Structure element requested. If the I-Structure element is in the "empty" or "deferred" state, then the request is added to the end of the deferred-reads queue and no further action is taken. Finally, as soon the I-Structure element becomes available, the value is sent back to the requester by using another *MPI_Send* call.

- `double DIS_RecvDatum(int index)`. An *MPI_Recv* instruction is executed to receive a message from `node`. `Index` specifies the position from the list of remote requests where the continuation vector is stored. This routine returns the value of the I-Structure element requested.
- `Void DIS_Write(int id, int pos, double value)`. This instruction stores a `value` in the I-Structure `id` at position `pos`. If that element is in the “deferred” state the value stored is copied to all continuation vectors and state is changed to “full”; if element is in “empty” state the value is stored in that position and its state is changed to “full”. If element is “full” state then the store operation cannot be completed and it causes a fatal error.

```
int main(int argc, char **argv ){
-CODE-
if (rank==0){
  for (j=1; j<PROCS; j++)
   for (i=0; i<n; i++)
    index[i]=DIS_Request(j, ID, i);
   for (j=1; j<PROCS; j++)
    for (i=0; i<n; i++)
     data[i]=DIS_RecvDatum(index[i]);
 }
 else{
  for (i=0; i<n; i++)
    DIS_Write(ID, i, value[i]);
  for (i=0; i<n; i++)
    DIS_RecvRequest(0);
 }
-CODE-
}
```

Fig. 3. Original user code

3.2 Code Transformation Description Following an Example Code

An example of code transformation by exploiting constant information is presented next. Fig. 3 shows the original user code to be optimized. In this code, process 0 sends request for *n* elements to the rest of the processes in the communicator.

As constant input information, we provide the following parameters: PROCS=2, n=3, ID=3. For rank=1 we define I-Structure elements such as: ID=3, element=0, value=12.7 and ID=3, element=2, value=38.5

3.2.1 Main-Body Routine Code Duplication

In this step, the original main-body routine code is copied as many times as there are specified processes. The main-body routine code is substituted for a switch-case instruction that selects the appropriate code for each process. The code for a particular process is specified by the function `main_process_X`, where `X` is the rank number and it is an exact copy of the original main-body code. In Fig. 4, we see how this code transformation is done in the example code: a new main-body code is inserted (lines 1-12) and it contains a switch instruction where the variable rank has two possible options because it is intended to run with two processes. Also, two new functions have been inserted in the code, `main_process_0` (lines 14-32) and `main_process_1` (lines 34-52), these functions specify the code for each process.

```
 1  Int main(int argc, char **argv ){
 2  DIS_Init(&argc,&argv);
 3  DIS_GetProcessRank(&rank);
 4  switch(rank) {
 5       case 0: main_process_0();
 6               break;
 7       case 1: main_process_1();
 8               break;
 9  };
10  DIS_Finalize();
11  return 1;
12  }
13
14  int main_process_0(){
15  -CODE-
16  if (rank==0){
17   for (j=1; j<PROCS; j++)
18    for (i=0; i<n; i++)
19     index[i]=DIS_Request(j, ID, i);
20   for (j=1; j<PROCS; j++)
21    for (i=0; i<n; i++)
22     data[i]=DIS_RecvDatum(index[i]);
23  }
24  else{
25   for (i=0; i<n; i++)
26    DIS_Write(ID, i, value[i]);
27  for (i=0; i<n; i++)
28   DIS_RecvRequest(0);
29  }
30  -CODE-
31  return 1 ;
32  }
33
34  int main_process_1(){
35  -CODE-
36  if (rank==0){
37   for (j=1; j<PROCS; j++)
38    for (i=0; i<n; i++)
39     index[i]=DIS_Request(j, ID, i);
40   for (j=1; j<PROCS; j++)
41    for (i=0; i<n; i++)
42     data[i]=DIS_RecvDatum(index[i]);
43  }
44  else{
45   for (i=0; i<n; i++)
46    DIS_Write(ID, i, value[i]);
47   for (i=0; i<n; i++)
48    DIS_RecvRequest(0);
49  }
50  -CODE-
51  Return 1 ;
52  }
```

Fig. 4. Main-body routine code duplication

```
 1  Int main(int argc, char **argv ){
 2  DIS_Init(&argc,&argv);
 3  DIS_GetProcessRank(&rank);
 4  switch(rank) {
 5       case 0: main_process_0();
 6               break;
 7       case 1: main_process_1();
 8               break;
 9  };
10  DIS_Finalize();
11  return 1;
12  }
13
14  int main_process_0(){
15  -CODE-
16  if (0==0){
17  for (j=1; j<2; j++)
18   for (i=0; i<3; i++)
19    index[i]=DIS_Request(j, 3, i);
20   for (j=1; j<2; j++)
21    for (i=0; i<3; i++)
22    data[i]=DIS_RecvDatum(index[i]);
23   }
24   else{
25   for (i=0; i<3; i++)
26    DIS_Write(3, i, value[i]);
27  for (i=0; i<3; i++)
28   DIS_RecvRequest(0);
29  }
30  -CODE-
31  return 1;
32  }
33
34  int main_process_1(){
35  -CODE-
36  if (1==0){
37   for (j=1; j<2; j++)
38    for (i=0; i<3; i++)
39    index[i]=DIS_Request(j, 3, i);
40   for (j=1; j<2; j++)
41    for (i=0; i<3; i++)
42    data[i]=DIS_RecvDatum(index[i]);
43  }
44  else{
45   for (i=0; i<3; i++)
46    DIS_Write(3, i, value[i]);
47   for (i=0; i<3; i++)
48    DIS_RecvRequest(0);
49   }
50  -CODE-
51  return 1;
52  }
```

Fig. 5. Constant propagation to identify static loops

3.2.2 Constant Propagation

In this step, we propagate constant information throughout the code to detect any possible static loop. In the example (see Fig. 5 for details), we propagate for rank=0 the constants PROCS=2, n=3, ID=3 and for rank=1 we propagate: PROCS=2, n=3, ID=3.

3.2.3 Dead-Code Elimination

Instructions that will never be processed by a particular process are eliminated in this step (see Fig. 6 for resulting code); for instance, conditional expressions depending on the rank value. In the example, from Fig. 5, we see that lines 24-29 in function `main_process_0` will never be executed by process 0; the same happens in function `main_process_1` where lines 36-43 will never be processed by process 1.

3.2.4 Unrolling Loops

All loops involving memory accesses are unrolled to detect possible static instructions inside loops. In the example code (Fig. 6), there are six loops that can be unrolled (lines 14, 15, 17, 18, 26, and 28). Fig. 7 shows the code after the loops have been unrolled.

```
 1  int main(int argc, char **argv ){
 2  DIS_Init(&argc,&argv);
 3  DIS_GetProcessRank(&rank);
 4  switch(rank) {
 5    case 0: main_process_0(); break;
 6    case 1: main_process_1(); break;
 7  };
 8  DIS_Finalize();
 9  return 1;
10  }
11
12  int main_process_0(){
13  -CODE-
14  for (j=1; j<2; j++)
15   for (i=0; i<3; i++)
16   index[i]=DIS_Request(j, 3, i);
17  for (j=1; j<2; j++)
18   for (i=0; i<3; i++)
19    data[i]=DIS_RecvDatum(index[i]);
20  -CODE-
21  return 1;
22  }
23
24  int main_process_1(){
25  -CODE-
26  for (i=0; i<3; i++)
27   DIS_Write(3, i, value[i]);
28  for (i=0; i<3; i++)
29   DIS_RecvRequest(0);
30  -CODE-
31  return 1;
32  }
```

Fig. 6. Code after dead-code elimination

```
 1  int main(int argc, char **argv ){
 2  DIS_Init(&argc,&argv);
 3  DIS_GetProcessRank(&rank);
 4  switch(rank) {
 5       case 0: main_process_0();
 6               break;
 7       case 1: main_process_1();
 8               break;
 9  };
10  DIS_Finalize();
11  return 1;
12  }
13
14  int main_process_0(){
15  -CODE-
16  index[0]=DIS_Request(1, 3, 0);
17  index[1]=DIS_Request(1, 3, 1);
18  index[2]=DIS_Request(1, 3, 2);
19  data[0]=DIS_RecvDatum(index[0]);
20  data[1]=DIS_RecvDatum(index[1]);
21  data[2]=DIS_RecvDatum(index[2]);
22  -CODE-
23  return 1;
24  }
25
26  int main_process_1(){
27  -CODE-
28  DIS_Write(3, 0, value[0]);
29  DIS_Write(3, 1, value[1]);
30  DIS_Write(3, 2, value[2]);
31  DIS_RecvRequest(0);
32  DIS_RecvRequest(0);
33  DIS_RecvRequest(0);
34  -CODE-
35  return 1;
36  }
```

Fig. 7. Unroll loops inside each local_main functions

3.2.5 Final Constant Propagation

We propagate constants throughout the code to reach variables inside the loops that may not be processed during first propagation. In Fig. 8, we show the code after propagation; lines 28 and 30 have been modified specifying the values to be stored in the I-Structure 3 positions 0 and 2.

3.2.6 Constant Requests Evaluation for Remote I-Structure Elements

This step detects static memory accesses and eliminates them. Each constant request is erased from the code and a `DIS_RemoteRequest()` function is inserted instead in the `main_process_X()` function of the remote process code.

```
 1  int main(int argc, char **argv ){
 2  DIS_Init(&argc,&argv);
 3  DIS_GetProcessRank(&rank);
 4  switch(rank) {
 5       case 0: main_process_0();
 6               break;
 7       case 1: main_process_1();
 8               break;
 9  };
10  DIS_Finalize();
11  return 1;
12  }
13
14  int main_process_0(){
15  -CODE-
16  index[0]=DIS_Request(1, 3, 0);
17  index[1]=DIS_Request(1, 3, 1);
18  index[2]=DIS_Request(1, 3, 2);
19  Data[0]=DIS_RecvDatum(index[0]);
20  data[1]=DIS_RecvDatum(index[1]);
21  data[2]=DIS_RecvDatum(index[2]);
22  -CODE-
23  return 1;
24  }
25
26  int main_process_1(){
27  -CODE-
28  DIS_Write(3, 0, 12.7);
29  DIS_Write(3, 1, value[1]);
30  DIS_Write(3, 2, 38.5);
31  DIS_RecvRequest(0);
32  DIS_RecvRequest(0);
33  DIS_RecvRequest(0);
34  -CODE-
35  return 1;
36  }
```

Fig. 8. Code after constant propagation

```
 1  int main(int argc, char **argv ){
 2    DIS_Init(&argc,&argv);
 3    DIS_GetProcessRank(&rank);
 4    switch(rank) {
 5      case 0: main_process_0();
 6              break;
 7      case 1: main_process_1();
 8              break;
 9    };
10    DIS_Finalize();
11    return 1;
12  }
13
14
15  int main_process_0(){
16    -CODE-
17    data[0]=DIS_RecvDatum(index[0]);
18    data[1]=DIS_RecvDatum(index[1]);
19    data[2]=DIS_RecvDatum(index[2]);
20    -CODE-
21    return 1;
22  }
23
24  int main_process_1(){
25    -CODE-
26    base=0;
27    DIS_Write(3, 0, 12.7);
28    DIS_Write(3, 1, value[1]);
29    DIS_Write(3, 2, 38.5);
30    DIS_RemoteRequest(0,3,0,base+0);
31    DIS_RemoteRequest(0,3,1,base+1);
32    DIS_RemoteRequest(0,3,2,base+2);
33    base=3;
34    -CODE-
35    return 1;
36  }
```

Fig. 9. Static messages evaluation by inserting `DIS_RemoteRequest()`functions in the data-owner (process that stores data) text code

The introduction of the `DIS_RemoteRequest()` functions insert in local I-Structure elements a remote deferred read. From Fig. 8, lines 16-18 are constant requests and can be transformed into `DIS_RemoteRequest()` functions as can be seen in Fig. 9 in lines 30-32. `Base` is a variable that adjusts index when loops involving memory requests cannot be unrolled.

3.2.7 Constant Remote Request Completion

In this step, each `main_process_X()`function is analyzed to check if any of the `DIS_RemoteRequest()`functions refers to an I-Structure element already defined by a `DIS_Write()` function. If so, there is no need to wait until execution time to complete this evaluation, it can be evaluated during this optimization step. Then, the corresponding `DIS_RecvDatum()`function can be deleted and substituted by the constant value already defined. From Fig. 9, lines 30 and 32 refer to an I-Structure element already defined in lines 27 and 29 respectively.

Therefore, lines 30 and 32 (Fig. 9) can be evaluated by copying values 12.7 and 38.5 into the `main_process_0()` code as is shown in Fig. 10, lines 17 and 19.

In this section, we have shown above how to partially evaluate remote memory requests by exploiting the I-Structures' features and constant propagation prior to the execution of the parallel program. In this particular data independent example, three of the messages needed to perform remote memory requests can be fully evaluated while 2/3 of the messages that answer remote requests can be also fully evaluated. Hence, from six messages that were required to be evaluated at execution time, five of them were evaluated during the optimization technique.

```
int main(int argc, char **argv ){
  DIS_Init(&argc,&argv);
  DIS_GetProcessRank(&rank);
  switch(rank) {
    case 0: main_process_0();
            break;
    case 1: main_process_1();
            break;
  };
  DIS_Finalize();
  return 1;
}


int main_process_0(){
  -CODE-
  data[0]=12.7;
  data[1]=DIS_RecvDatum(index[1]);
  data[2]=38.5;
  -CODE-
  return 1;
}

int main_process_1(){
  base=0;
  -CODE-
  DIS_Write(3, 0, 12.7);
  DIS_Write(3, 1, value[1]);
  DIS_Write(3, 2, 38.5);
  DIS_RemoteRequest(0,3,1,base+1);
  base=3;
  -CODE-
  return 1;
}
```

Fig. 10. Constant information in remote node is transferred to the requester

4 Experimental Results

This optimization technique has been tested with several algorithms such as matrix multiplication, conjugate gradient, and fast Fourier transform [7, 8] running in a SGI Origin 2000 with 10 MIPS R10000 processors and a PC Cluster with 8 Pentium III processors. In this section, we show experimental results for a 4 Dual-Pentium III PC Cluster in a 10/100 Fast Ethernet point-to-point interconnection and 512 MB of memory in each node. Programs presented in the section use no collective communication, cache mechanism, message coalescing, or data locality exploitation. These restrictions are set just to observe how much performance can be obtained just by the partial evaluation technique alone.

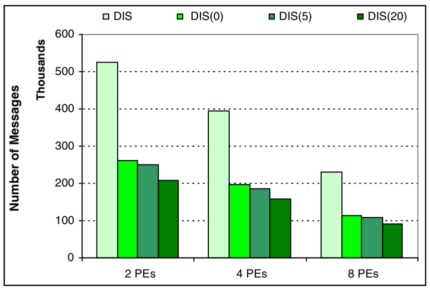
We present experimental results using the 2D Haar wavelet transform (2D-HWT) applied to a 1024x1024 image. The Haar wavelet transform is the first known wavelet, proposed in 1909 by Alfred Haar [17]. The Haar wavelet is also the simplest possible wavelet. As opposed to the functions sine and cosine used for Fourier transforms, a wavelet not only has locality in the frequency domain but also in the time or spatial domain. The algorithm produces as output a file containing the average of original image together with the detail information of the same image.

We chose 2D-HWT because it is a data independent algorithm. This feature makes it well suitable to show the advantages of our optimization technique. With this benchmark program, we intend to demonstrate how parallel programs can benefit when part of the input information is constant. In benchmark program, we assume that different percentages of the input image are known. This assumption is reasonable in digital image processing where images may contain a constant background or fixed objects. In experiments, we run the program that implements 2D-HWT and use D-IS memory system.

We show results for different percentages of the image, network latencies, and number of processing elements (PEs). We define the following notation:

DIS - Refers to the original program without any optimization.

DIS(p) - Refers to the optimized program running when *p* percentage of the image is known. When zero percentage of the image is known, technique can still be performed because the sending of requests can be evaluated if image size is provided.



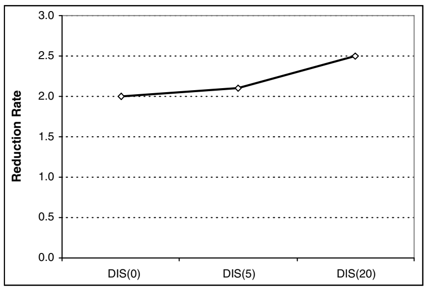


Fig. 11. Number of messages sent varying the number of processing elements and the percentage of the image that is known

Fig. 12. Reduction in the message rate when part of the image (0%, 5%, and 20%) is known

4.1 Number of Messages Analysis

Fig. 11 shows how the number of messages sent by optimized and non-optimized programs varies with respect to the number of PEs. Comparing *DIS* and *DIS(0)* from this figure, we can see that optimization technique can eliminate half of the messages just by knowing the image dimension and the number of processing elements available. Under these circumstances memory requests can be sent even without knowing the value of any pixels of the image.

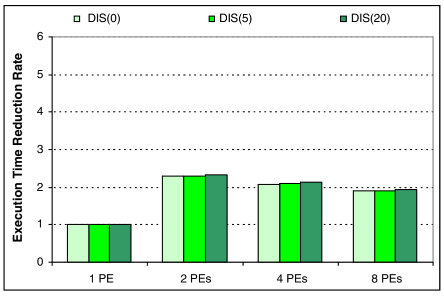
These instructions represent half of the messages to send; the other half is required to send the value of elements when they become available.

We also see that the number of messages is reduced when the number of processing elements increases; this is an effect of parallelization and data distribution. Comparing *DIS(0)*, *DIS(5)* and *DIS(20)*; we also see the impact of the technique when part of the image is known. In this case, not only the requests can be performed which is the case between *DIS* and *DIS(0)*, but also some requests can be answered, thereby eliminating more messages, as seen in Fig. 11.

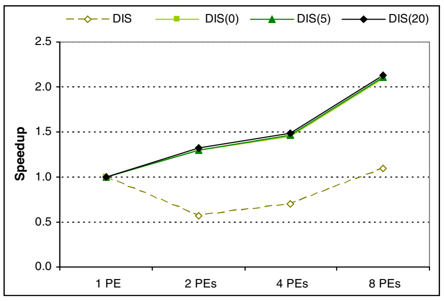
These results are confirmed in Fig. 12, which shows the reduction in the rate of message. This measurement is the ratio between the number of messages sent by the *DIS* program over the number of messages sent by the *DIS(k)* programs. As seen in the figure, this ratio is at least two and increases when part of the image is known. This happens for 2, 4, and 8 PEs.

4.2 Execution Time Reduction Analysis

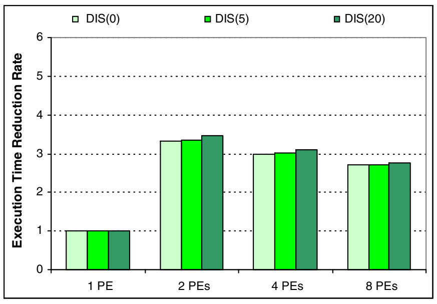
Fig. 13 shows the execution time reduction rate obtained with *DIS* program varying the percentage of constant information, number of PEs and the interconnection network latency. Execution time reduction rate is the ratio between *DIS* execution time over *DIS(k)* execution time. From this figure, we see the impact of the technique with different interconnection network latencies.



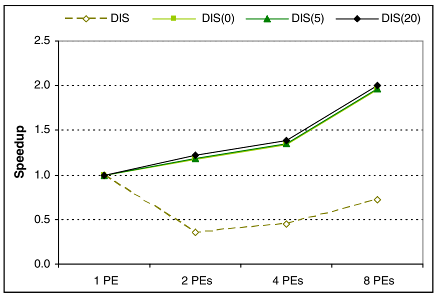
(a)



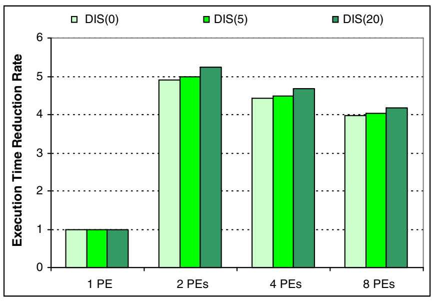
(a)



(b)

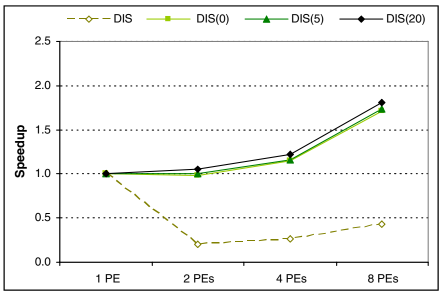


(b)



(c)

Fig. 13. Execution time reduction rate varying the number of PEs, the percentage of constant information and the interconnection network speed. We analyze (a) twice faster (b) original and (c) twice slower network speeds



(c)

Fig. 14. Speedup of *DIS, DIS(0), DIS(5), DIS(20)* programs with different numbers of PEs. We present data for (a) twice faster (b) original and (c) twice slower interconnection network

In Fig. 13a, the interconnection network is twice faster than network in Fig. 13b and four times faster than in Fig. 13c; while the network in Fig. 13b is twice faster than the network in Fig. 13c. Hence, from this figure we see that the reduction rate is higher when the interconnection network is slower. This means that technique makes single assignment memory system more robust and latency tolerant.

We also see that there is almost no optimization possible when there is just one PE because technique gets its real advantage from remote memory operations instead of local memory operations. Also, when we increase the percentage of constant input information from 0, 5, and 20, there is a small increment in the reduction ratio because a second message is eliminated; however, the processing of that message is not so time-consuming when compared with the time spent by sending and receiving requests. Moreover, optimization is reduced when the number of PEs is increased. This is due to the data distribution between PEs; in other words, when more PEs are added, then more messages are required to exchange information.

This effect does not mean that the optimized program runs slower; this only means that the original program execution time and the execution time of its optimized version are becoming similar.

4.3 Speedup

Fig. 14 shows the speedup obtained by benchmark programs when increasing the number of PEs and varying the interconnection network speed by a factor of two. We compare the time spent by parallel programs running in several PEs with respect to the same parallel implementation running in a single PE.

Fig. 14a, 14b, and 14c show that *DIS* programs have a speedup below one which means that programs with more than one PE run slower than their sequential counterpart. This is due to the exchange of messages, which are time consuming; however, with the introduction of more PEs, the program begins speeding up. When the interconnection network is fast enough, the speedup becomes higher than one (see Fig. 14a *DIS* with 8 PEs). However, when the technique is applied to *DIS* program even without any image values, which is the case of *DIS(0)*, we note a positive speedup. This tendency is also valid for *DIS(5)* and *DIS(20)* execution times.

In these cases, the overhead introduced by the management of I-Structures and the communication times can be masked by the technique, producing a faster optimized code. *DIS(0), DIS(5),* and *DIS(20)* display a similar speedup because the execution time is similar in these cases.

5 Related Work

In this section we review related work in the area of parallel program optimization. We analyze optimizations performed to the communication library (MPI) in software and hardware also we review optimizations performed at compiled time which exploits static information about network or communication patterns.

5.1 Optimization of Inter-Process Communication

Optimizations of the MPI barrier operation are discussed in [19]. Moh *et al* propose a fast tree-based barrier synchronization scheme for 2-D meshes producing a reduction in the number of messages by combining the synchronization messages.

In [4], a design and implementation of the MPI collective communication instructions optimized for clusters of workstations is presented. The system consists of two main components: the MPI-CCL layer and a User-Level Reliable Transport Protocol (URTP). The MPI-CCL layer includes the collective communication functionality of MPI and the URTP works as an interface with the LAN Data-Link Layer. Their system is integrated with the operative system through a kernel extension mechanism. These operations reduce significantly the number of messages during the execution of a MPI program. However, the correct utilization of these instructions depends on the ability of programmer.

In [10], a prototype of the D-OSC, a SISAL compiler for distributed memory machines is presented. D-OSC is a further research of the Optimizing SISAL Compiler (OSC) [16]. D-OSC generates C code with MPI calls. In D-OSC, messages are eliminated using rectangular arrays, multiple-alignment, and block messages.

In [13], a library of collective communication operations, called MAGPIE, is presented. MAGPIE is optimized for wide area systems and its algorithms are designed to send the minimal amount of data over the slow wide area links, and to only incur singlewide area latency. MAGPIE implements the complete set of collective operations according to the MPI standard. Reduction operations with short data vectors are frequently used in parallel applications. The paper also discusses optimizations such as message vectorization, message coalescing, and redundancy elimination implemented in MAGPIE.

5.2 Optimizations at Compile-Time

Single assignment is a fundamental property of variables in functional languages. When a variable is only assigned to a value once, then an instance of that variable is thereafter semantically equivalent to the value. The single assignment property is used in compilers to implement a variety of optimizations [5]. One of the most attractive features of single-assignment in parallel systems is that cache coherence is already embedded in it [15].

The PARADIGM compiler [3], provides an automated mean to parallelize sequential programs for their efficient execution on distributed-memory multi-computers. PARADIGM performs a number of optimizations: automatic data partitioning and distribution, synthesis of high-level communication, and communication optimization. These are provided through a generic library interface (MPI is included). Regular computations are optimized by message coalescing, message vectorization, coarse grain pipelining, and message aggregation. It also supports functional, data parallelism, and multithreaded execution.

In [1], a compiler algorithm that automatically finds computation- and data-decompositions is presented. This algorithm optimizes both parallelism and data locality. Also, a mathematical framework to systematically derive decompositions is introduced. An optimization algorithm focuses on programs with nesting of parallel and sequential loops. The algorithm attempts to uncover a static decomposition that exploits the maximum degree of parallelism available in the program to minimize communication, such that there is no reorganization or pipeline communication. It can exploit parallelism in both fully parallelizable loops as well as loops that require explicit synchronization. If communication is needed, the algorithm will attempt to introduce the least expensive forms of communication into those parts of the program that are least frequently executed.

Another optimization technique performed at compile-time and applied to message-passing parallel programs is Compiled Communication (CC) [21]. In CC, the compiler determines the communication requirements in a program and manages network resources, such as multicast groups and buffer memory, statically using the knowledge of both the underlying network architecture and the application communication requirement. In this technique, the compiler analyzes the program and partitions it into phases. Each phase has a fixed communication pattern and the compiler inserts code to reconfigure the network at the end of each phase to manage network resources directly. CC can eliminate runtime communication overhead produced by group management. CC can also use prolonged connections for communications and amortize the startup overhead over a number of messages. However, CC cannot be applied to communications where information is not available at compile time. In other words, the programming style influences the effectiveness of the CC technique. Recently, CC has been proposed to improve the performance of MPI routines for clusters of workstations, and an MPI prototype called CC-MPI [12] has been designed. The CC-MPI supports compiled communication on Ethernet switched clusters. It allows the user to manage network resources such as multicast groups directly and to optimize communications based on the availability of the communication information. The CC-MPI optimizes one-to-all, one-to-many, all-to-all, and many-to-many collective communication routines using the CC technique.

6 Conclusions

In this paper, we have provided detailed information about how to perform code manipulations in order to optimize parallel programs by exploiting static information. This technique eliminates messages if the input data of MPI_Send() and MPI_Recv() routines are known. We show that code transformations can be considered as efficient optimization tool and they can be done by a partial evaluator using D-IS memory system. We have shown that partial evaluation can be extended to a wider class of program paradigms, and efficiently applied to distributed-applications, reducing the number of the most time-consuming operations in addition to the known optimizations of sequential programs. In some applications with a partially given input, the number of remote memory requests can be decreased dramatically by evaluating ready-to-execute MPI_Send() and MPI_Recv() routines. Traffic in the interconnection network and the network latency is also reduced and it makes the system more scalable especially with slow interconnection media.

Technique also improves design process avoiding hand-made optimization and exploiting features of parallel system automatically. Technique may also increase code and memory consumption while improving efficiency; however, the same occurs with traditional partial evaluation technique. The regulation of extra code inserted is made by limitation of unfolding or depth of recursion, or loop unrolling, etc. automatically or with human interaction during partial evaluation step. Moreover, code that handles transactions (send/receive routines) could grow with less speed than specialized code for each processor. In any case, elimination of the messages is much more time saving than time increasing by code growing.

Acknowledgements. The authors are pleased to acknowledge the anonymous referees whose valuable remarks and comments helped improve the paper. This work is supported in part by the UC-Mexus under 2003-2004 post doctorate program, by UABC under grant #371, by SEP-PROMEP under grant #PTC-UABC-16, by the NSF under Grants #CCR-0234444 and INT-0223647, and by CONACYT under grant #32989. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the UC-Mexus, UABC, SEP, NSF, or of CONACYT.

References

1. Amarasinghe S-P and Lam M-S. Communication optimization and code generation for distributed memory machines. In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation. 1993.
2. Arvind, Nikhil R-S, Pingali K-K. I-Structures: Data Structures for Parallel Computing. ACM Transaction on PLS, Vol. 11 No. 4 pp. 598-632. 1989.
3. Banerjee P., Chandy J-A, Gupta M., Holm J-G, Lain A, Palermo D-J, Ramaswamy S., Su E. The PARADIGM compiler for distributed-memory message multicomputers. In proceedings of the first international workshop on parallel processing. 1994.
4. Bruck J., Dolev D., Ho C-T, Roşu M-C, Strong R. Efficient Message-passing Interface (MPI) for Parallel Computing on Clusters of Workstations. Journal of Parallel and Distributed Computing, Vol. 40 No. 1 pp. 19-34. 1997.
5. Champeaux D., Lea D., and Faure P. Object-Oriented System Development. Addison Wesley, ISBN 0-201-56355-X. 1993.
6. Cristóbal-Salas A, and Tchernykh A. I-Structure Software Cache for distributed applications. Dyna, Year 71, No. 141. pp. 67 – 74. Medellín, March 2004. ISSN 0012-7353. 2004
7. Cristóbal-Salas A., Tchernykh A., Gaudiot J-L., Lin WY. Non-Strict Execution in Parallel and Distributed Computing, International Journal of Parallel Programming, Kluwer Academic Publishers, New York, U.S.A., Vol. 31, 2, p. 77-105. 2003.
8. Cristóbal-Salas A., Tchernykh A., Gaudiot J-L. Incomplete Information Processing for Optimization of Distributed Applications. Proceedings of the Fourth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03), ACIS, pp. 277-284, 2003.
9. Faraj A-A. Communication characteristics in the NAS parallel benchmarks. Master thesis, college of arts and sciences, Florida State University. October 2002.
10. Garza-Salazar D-A, Bohm W. D-OSC: A sisal compiler for distributed memory machines. In proceedings of the International Workshop on PCS. 1997.
11. Jones, N-D. An introduction to Partial Evaluation. ACM computing surveys, Vol. 28, No. 3. 1996.

12. Karwande A., Yuan X., and Lowenthal D-K. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 95-106. 2003.
13. Kielmann T., Hofman F-H, Bal H-E, Plaat A., and Bhoedjang A-F. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems, 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99). 1999.
14. Lahaut D. and Germain C. Static Communications in Parallel Scientic Programs. In PARLE'94, Parallel Architec-ture & Languages, LNCS 817, pp. 262-276. 1994.
15. Lin W-Y, and Gaudiot J-L. 1996. I-Structure Software Cache - A split-Phase Transaction runtime cache system, Proceedings of PACT '96 Boston, MA. 1996.
16. McGraw J., Skedzielewski S., Allan S., Grit D., Oldehoeft R., Glauert J., Dobes I., and Hohensee P. SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual, version 1. 2. Technical Report TR M-146, University of California - Lawrence Livermore Laboratory. 1985.
17. Mikulic Emil. Haar wavelet transform. http://dmr.ath.cx/gfx/haar/index.html. 2004.
18. Mogensen and P Sestoft. Partial evaluation. In A. Kent and J.G. Williams, editors, Encyclopedia of Computer Science and Technology, Vol. 37, pp. 247-279. 1997.
19. Moh S., Yu C., Lee B., Youn H-Y, Han D., Lee D. 4-ary Tree-Based Barrier Synchronization for 2-D Meshes without Nonmember Involvement. IEEE Transactions on Computers, Vol. 50, No. 8. 2001.
20. Ogawa H., Matsuoka S. OMPI: Optimizing MPI programs using Partial Evaluation. Proceedings of the 1996 IEEE/ACM Supercomputing Conference, Pittsburgh. 1996.
21. Yuan X., Melhem R. and Gupta R., Algorithms for Supporting Compiled Communication. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 2, pp. 107-118. 2003.