Kandó Kalman College
Institute of Computer Technology
Székesfehérvár
Hungary

HPCL
Computer Technology Institute
Athens
Greece

TEMPUS SJEP N° 08353-94

Activity N° 6.

Digital Signal Processing

Realization of Parallel DSP Algorithms on the Wavefront Array Processors

Dr. Seebauer Marta,
Prof. Sergios Theodoridis, Nastou Panagiotis

Athens - 1995

Introduction

Purpose of the mobility was to realise a 8 weeks mobility program for implementation of selected DSP algorithms, with parallel processing in High Performance Computer Laboratory (HPCL) of the Computer Technology Institute (CTI) Athens under Activity N^{o} 6. Digital Signal Processing of the TEMPUS SJEP N^{o} 08353-94.

Athens High Performance Computing Laboratory has been established in March 1993 with the cooperation of National University of Athens, National Technical University of Athens and Computer Technology Institute.

One of its main principal objective is to provide and design better and more efficient models, algorithms and interconnection systems for better exploitation of parallel systems. The HPCL of Computer Technology Institute Athens has high-qualified staff members in the fields of parallel and signal processing.

The in this work realised implementation examples is the practical piece of the chapter Realisation of Parallel DSP Algorithms on the Wavefront Array Processors for the syllabus of course Digital Signal Processing.

My thanks are due to prof. Sergios Theodoridis for ideas and Nastou Panagiotis for his patience.

1. Hardware and Software Environment

The PARSYTEC GCel 3/512 is a MIMD supercomputer with 512 processors. That is a machine of 8 cubes, each containing 64 processors, currently transputers T805.

The host computers are 2 SUN workstations as universal gateways between the outer world and the parallel computing devices driven by PARIX. One of them has 8 links, and the other one 4 links for the communication with the transputer system. The primary task of the host computers is to supply the user with a development environment and to give parallel applications first access to external facilities. Hosts are bridged to the individual partitions of the transputer system's "bus bridgeheads". These interface modules - occupying at least one slot of the host's external bus connectors - allow data exchange between the hosts and the multi-processor network and convert serial link protocols into parallel bus-accesses.

The PARSYTEC GCel equipped with T805 transputers is offered as an entry level GC system. Using a similar infrastructure this massively parallel system can easily be upgraded to a GC system with T9000 transputers.

To enable a smooth migration from existing systems towards T9000 PARIX-based systems, the PARIX User Environment including the ACE EXPERT compilers are completely available for current T800 based system. The concept, however, is PARSYTEC GC oriented - and thus provides required T9000 functionalities like virtual channels, virtual links, semaphores, and extended debugging features also for the T800/T805. An additional kernel layer emulates some features of the T9000/C104, which are not supported in hardware on the T80x. These are mainly the packetized communication protocols and routing facilities, which are based on virtual links.

PARIX as the software environment for PARSYTEC GC series has to serve the specific needs of this massively parallel architecture targetting towards highest performance for innovative applications. PARIX is based on UNIX as the frontend operating system, with parallel extensions catering for the needs of the parallel supercomputer. To a great extent PARIX is hardware independent.

PARIX as a distributed operating environment puts together software for machine administration and for application development and execution by the user. Some components of PARIX reside on the frontend machine, while others are running on the transputer system itself.

Easily to understand the application examples let see the PARIX programming model which is designed complying with existing and future requirements for parallel algorithms. It incorporets a variety of static and fully dynamic programming features and is an easy-to-use approach for a large set of applications:

Static facilities:
- identical main program loaded onto all processors,
- identification of the individual position within the network,
- pre-established topologies based on virtual links.

Dynamic facilities:
- creating individual light-weight processes (thread),
- creation of virtual links between arbitrary processes,
- synchronous and asynchronous communication,
- handling of user-defined virtual topologies,
- load and execute of additional code (context).

External services:
- host access by using Remote Procedure Calls (RPC),
- access to external servers via RPCs.

Other conceptual highlights:
- implicit process graph modelling, no network description language,
- no centralized software instances and services.

Typical program development cycle for PARIX applications:

. A parallel algorithm is modelled and the application itself is conceptually targetted towards the PARIX programming model or one of the third party programming models residing on top of PARIX. This usually can be done without having access to any kind of parallel processing resources.

. Then the different components of the application are coded by using a suitable editor program under UNIX and compiled by running one of the ACE EXPERT compilers for the transputer.

. After the compilation and linkage step are successfull completed, the program may be downloaded and run on a suitable transputer network. The PARIX run-time system is established on the transputers at load time and finishis its work at termination point.

. The next step usually contains functional debugging where potencial lacks within the algorithm and its implementation may be detected (dead-locks etc.). This is facilitated by the PARIX source- and assembly- level debugging tools.

. The final step incorporetes performance optimization to get the best utilization of the underlying hardware resources. Besides individual local measurement on the base of transputer timers, more data may be generated by using performance analyzing tools as PATOP.

Compilers are available for C, ANSI C, and FORTRAN 77. Extensions for parallel processing, communication and host access are realized via library calls for all languages.

Besides the PARIX kernel, wich supplies the basic services of PARIX several libraries are supplied to statisfy the programmer's demands. This libraries cover the different areas of computing, such as I/O, numeric, graphics and communication. Where possible, the functions are realized on the transputer node.

Many applications require regular communication structure, like pipe, ring, grid or tree. These sets of links are managed with the PARIX virtual topology facility. To achieve optimal throughput and convenient application modeling, the communication networks need to be mapped optimally onto the physical processor and link network. Inadequate creation of virtual topologies can have severe effects on the overall performance of an application. To support efficient mapping of topologies, PARIX contains topology libraries. They map some of the most common communication structures onto a physical connections, create the necessary virtual links and gather them virtual topologies. Calls like MAKE_PIPE or MAKE_TREE establish the necessary communications links in an optimized way and provide information about the caller's position in the topology's communication network.

2. Typical Topologies for the Parallel DSP Algorithms

The DSP algorithms (convolution, filtering, Fourier transform, matrix manipulations) are easily to implement on a pipeline based topology. A parallel pipeline system is one in which every data element is passed through every processor with each processor performing a different operation on each of the data elements. This approach is sometimes called algorithmic parallelism because the algorithm tasks are distributed among multiple processors. In contrast, data parallelism is a method in which the entire program is placed in every processor and data is distributed among the processors.

To use a multitransputer system is seemingly the case of data parallelism, too, while the entire program is loaded in each processor element. On other hand each processor element have to establish his own position and to execute the only commands which are dedicated to him, therefore it is algorithmic parallelism.

In general the pipeline methods tend to be fairly inflexible because the data must pass sequentially through a set of processors with specified operations performed on the data in each processor within the pipeline. Because the data moves from

one processor to the next, no processor can proceed until the previous processor in the pipe has completed its task and passed the data to it. This means that if one processor has less work than another, the faster processor will simply have to wait for the slower. In order to guarantee efficient use of processors in a pipeline, every processor in the pipeline must have exactly the same amount of work.

The parallel DSP algorithms are easily to divide on the same quantity of steps. The number of available in the system processor nodes would determine the number of parallel processes, however the feature of transputers to start more threads in one processor in the same time allows the facility to increase the number of parallel process. By PARSYTEC GCel the recommended maximal number of thread in one processor is 16. It is obvious that the use of many threads in every processor complicates the solution of the equivalent load of processor elements.

Beyond program distribution, the efficiency of a pipe will depend on the number of processor in the pipe in relation to the amount of data to be processed, and the efficiency with which the processors can communicate.

There are four types of communication available in PARIX, two of them are used for the communication on virtual links between the processor elements:

. *Synchronous virtual link*
  bound communication is the transputer's native form of communication. The communicating processes have to be connected with a (virtual) link and are synchronized upon communication. The process, which reaches the point of communication first has to wait for the other one, and data transfer takes place when both processes are ready to communicate. A process can await communication from one of several links simultaneously.

. *Asynchronous link*
  bound communication can be carried out concurrently with computation. It therefore optimizes processor usage, as sending and receiving is performed in the background while the processor continues working. The intermediate buffering is supported at both sender and receiver. On the other hand, the asynchronous link communication needs a longer timefor the communication through the buffering. The efficiency of its use depends on number of processor in the pipe and on work time of processor between the communication cycles.

It is necessary to note that the communication time significantly depends on the physical position of processor in the virtual topology and on its distance from the host.

The asset of transputer system is the facility to develop the most suitable hardware topology with use of software environment. It is in particular to prefer

the use of virtual topology like the optimal solution. If it needs the arbitrary links of virtual topology can be erased or any links added to topology.

For the convolution and MA filtering the most suitable topology is the one-dimensional pipe while the a part of data element has to travel systematical from the pipe_head to the pipe_tail.

By the AR and ARMA filtering the each pipe element has besides the access to the output data, too. In this case the most suitable topology is a one-level tree with the pipe_tail in the root and with the rest of pipe element in the leaves.

3. Realization of Parallel DSP Algorithms in C Programming Language

3.1. Convolution

The algorithm for the convolution of two signals

$$\begin{array}{ll} u(i) & i=\overline{0,N-1} \\ w(i) & i=\overline{0,M-1} \end{array} \quad \text{and}$$

is given in formula 3.1.

$$\begin{aligned} y(i) &= \sum_{j=0}^{i} u(j)w(i-j), \qquad if \quad i=\overline{0,M-1} \\ y(i) &= \sum_{j=i-N+1}^{N-1} u(j)w(i-j), \qquad if \quad i=\overline{M,N+M-2} \end{aligned} \tag{3.1}$$

The first realization of the program shows the the classical method of pipeline process, when it is supposed that the two signal sequences *u(i)* and *w(i)* are known at the time *i* yet.

```
/*  conv_clas.c

    Classical version of the
    program for realization of
          convolution
```

```
        of two signal sequences
                with
     synchronous communication
   between the processor elements

*/
#include <stdio.h>
#include <virt_top.h>
#include <sys/comm.h>
#include <sys/time.h>

#define     BACKWARD 0      /*virtual link from previous processor*/
#define     FORWARD  1      /*virtual link to next processor*/
#define     N       1000    /*number of input signal*/
#define     M       256     /*number of processor element*/

int main (void)
{
  int         first, second;
  double        work_time;

  int         i;
  int         u[N];
  int         w[M];
  int         y = 0;

  int         id;

  int         topId;
  int         reqId = 0;
  PipeData_t    *pipeData = NULL;

  int         pipe_len = M;

  topId = MakePipe(reqId, pipe_len, MINSLICE, MAXSLICE,
                MINSLICE, MAXSLICE, MINSLICE, MAXSLICE);
  pipeData = GetPipe_Data(topId);

  id = pipeData->id;

  /* assignment for signal sequences u[i] and w[i] */
  for (i = 0; i < N; ++i)
      u[i] = i;
  for (i = 0; i < pipe_len; ++i)
```

```
    w[i] = i;

/* convolution */

    switch(pipeData->status){
          case PIPE_NONE:
              break;

        case PIPE_HEAD:
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y = 0;
              else
                      y = w[id] * u[i];
                  Send(topId, FORWARD, &y, sizeof(int));
          }
              break;

        case PIPE_IN:
              for (i = 0; i < N+pipe_len-1; ++i){
                  Recv(topId, BACKWARD, &y, sizeof(int));
                  if (i < id || i > N+id-1 )
                      y = y + 0;
                  else
                      y = y + w[id] * u[i-id];
                  Send(topId, FORWARD, &y, sizeof(int));
              }
              break;

         case PIPE_TAIL:
              first = TimeNowHigh();
              for (i = 0; i < N+pipe_len-1; ++i){
                  Recv(topId, BACKWARD, &y, sizeof(int));
                  if (i < id)
                      y = y + 0;
                  else
                      y = y + w[id] * u[i-id];
                  printf("%6d\n", y);
               }
               second = TimeNowHigh();
               work_time = (double) (second-first)/ (double) CLK_TCK_HIGH;
               printf("\nWork time: %6.24f sec\n", work_time);
               break;
    }
```

```
FreePipe(topId);
return 0;
}
```

The liability of this solution is that all processor elements have to wait for the results of computing on previous processor element. A simple change of the order of commands and declaration of a new variable result a significant higher efficiency.

```
/*   conv_synch.c

    Program for realization of
          convolution
        of two signal sequences
        with
    synchronous communication
  between the processor elements

*/

#include <stdio.h>
#include <virt_top.h>
#include <sys/comm.h>
#include <sys/time.h>

#define     BACKWARD 0      /*virtual link from previous processor*/
#define     FORWARD  1      /*virtual link to next processor*/
#define     N       1000    /*number of input signal*/
#define     M       256     /*number of processor element*/

int main (void)
{
  int         first, second;
  double        work_time;

  int         i;
  int         u[N];
  int         w[M];
  int         y_send = 0;
  int         y_recv = 0;

  int         id;
```

```
int         topId;
int         reqId = 0;
int         prev;
PipeData_t    *pipeData = NULL;
int         pipe_len = M;

topId = MakePipe(reqId, pipe_len, MINSLICE, MAXSLICE,
                 MINSLICE, MAXSLICE, MINSLICE, MAXSLICE);
pipeData = GetPipe_Data(topId);

id = pipeData->id;

/* assignment for signal sequences u[i] and w[i] */

for (i = 0; i < N; ++i)
    u[i] = i;
for (i = 0; i < pipe_len; ++i)
    w[i] = i;

/* convolution */
  switch(pipeData->status){
          case PIPE_NONE:
              break;

        case PIPE_HEAD:
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y_send = 0;
              else
                      y_send = w[pipe_len-1-id] * u[i];
                  Send(topId, FORWARD, &y_send, sizeof(int));
            }
                  break;

        case PIPE_IN:
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y_send = y_recv + 0;
                  else
                      y_send = y_recv + w[pipe_len-1-id] * u[i];
                  Send(topId, FORWARD, &y_send, sizeof(int));
              Recv(topId, BACKWARD, &y_recv,sizeof(int));
            }
```

```
                break;

            case PIPE_TAIL:
                first = TimeNowHigh();          /*start timer*/
                for (i = 0; i < N+pipe_len-1; ++i){
                    if (i > N-1)
                        y_send = y_recv + 0;
                    else
                        y_send = y_recv + w[pipe_len-1-id] * u[i];
                    Recv(topId, BACKWARD, &y_recv, sizeof(int));
                    printf("%6d\n", y_send);      /*print result*/
                }
                 second = TimeNowHigh();          /*stop timer*/
               work_time = (double) (second-first)/CLK_TCK_HIGH;
                 printf("\nWork time: %6.24f mikrosec\n", work_time);
                 break;
  }
FreePipe(topId);
return 0;
}
```

Using synchronous method for communication between the processor elements only one pair of processor can occupy the virtual link at the same time. A signifficant improvement can be arrieved with the use of asynchronous method of communication. The liability of asynchronous communication are the longer communication's cycle and the use of memory place.

```
/*   conv_asynch.c

    Program for realization of
          convolution
      of two signal sequences
             with
    asynchronous communication
  between the processor elements

*/

#include <stdio.h>
#include <virt_top.h>
#include <sys/comm.h>
#include <sys/time.h>
```

```
#define     BACKWARD 0      /*virtual link from previous processor*/
#define     FORWARD  1      /*virtual link to next processor*/
#define     N       1000    /*number of input signal*/
#define     M       256     /*number of processor element*/

int main (void)
{
  int         first, second;
  double        work_time;

  int         i;
  int         u[N];
  int         w[M];
  int         y_send = 0;
  int         y_recv = 0;
  int         id;

  int         topId;
  int         reqId = 0;
  PipeData_t    *pipeData = NULL;

  int         pipe_len = M;
  int         error;

  topId = MakePipe(reqId, pipe_len, MINSLICE , MAXSLICE,
                MINSLICE, MAXSLICE, MINSLICE, MAXSLICE);
  AInit(topId, -1, -1);
  pipeData = GetPipe_Data(topId);

  id = pipeData->id;

  /* assignment for signal sequences u[i] and w[i] */
  for (i = 0; i < N; ++i)
      u[i] = i;
  for (i = 0; i < pipe_len; ++i)
      w[i] = i;

  /* convolution */

     switch(pipeData->status){
           case PIPE_NONE:
              break;
```

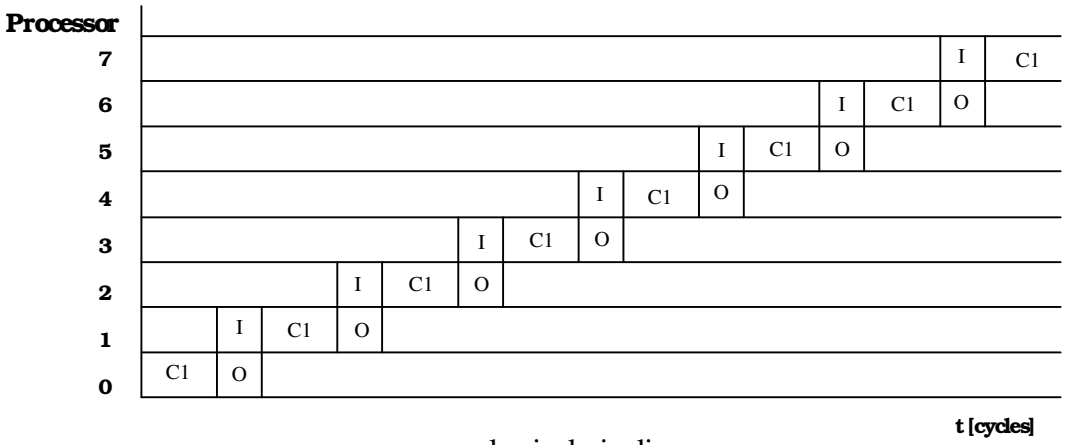
```
        case PIPE_HEAD:
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y_send = 0;
              else
                      y_send = w[pipe_len-1-id] * u[i];
                  ASend(topId, FORWARD, &y_send, sizeof(int), &error);
              }
              break;

        case PIPE_IN:
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y_send = y_recv + 0;
                  else
                      y_send = y_recv + w[pipe_len-1-id] * u[i];
                  ASend(topId, FORWARD, &y_send, sizeof(int), &error);
               ARecv(topId, BACKWARD, &y_recv, sizeof(int), &error);
                  ASync(topId, BACKWARD);
              }
              break;

         case PIPE_TAIL:
              first = TimeNowHigh();
              for (i = 0; i < N+pipe_len-1; ++i){
                  if (i > N-1)
                      y_send = y_recv + 0;
                  else
                      y_send = y_recv + w[pipe_len-1-id] * u[i];
                  ARecv(topId, BACKWARD, &y_recv, sizeof(int), &error);
                  ASync(topId, BACKWARD);
                  printf("%8d\n", y_send);
              }
              second = TimeNowHigh();
              work_time = (double) (second-first)/ (double) CLK_TCK_HIGH;
             printf("\nWork time: %6.24f sec\n", work_time);
             break;
    }
FreePipe(topId);
return 0;
}
```

To compare the efficiency of different realizations let see the results of run times of the above-mentoined programs. The run time was measured with use of on transputer own timer.

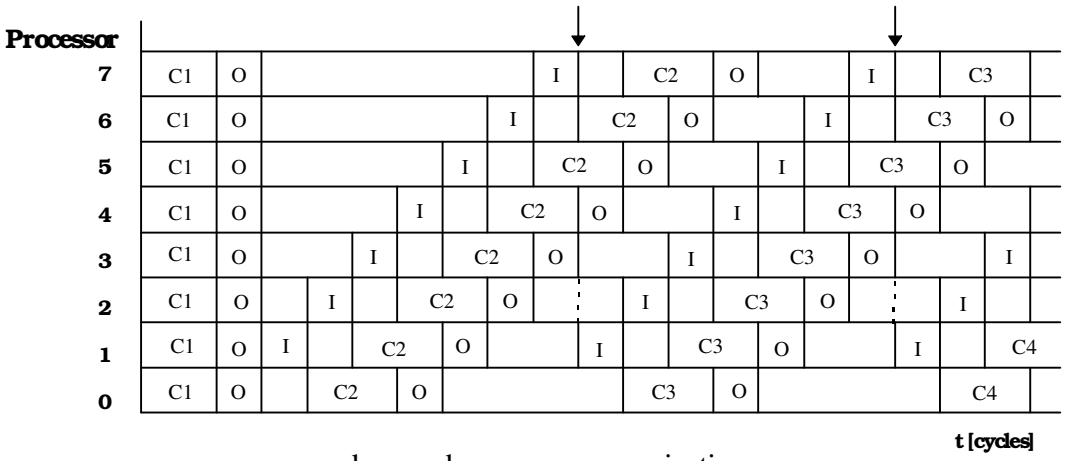
Run time results for programs of convolution t [sec]				
Signal length	Pipe length			
N	M=32	M=64	M=128	M=256
Classical pipeline				
100	0,687	0,828	1,277	4,125
200	1,142	1,329	2,095	8,146
500	2,692	4,235	4,624	13,856
1000	10,724	13,254	17,516	21,578
Synchronous communication				
100	0,462	2,152	1,464	5,406
200	1,198	2,503	1,855	8,494
500	4,359	6,146	4,843	14,223
1000	9,654	13,263	13,051	19,798
Asynchronous communication				
100	1,301	1,432	2,154	4,990
200	2,001	3,508	2,917	6,924
500	4,903	7,616	5,858	9,807
1000	6,466	9,324	10,691	16,522

From this data and from the next p**rocessor activity diagram** is to see that the efficiency of realization of a pipeline based parallel algorithm is depend on method of communication between the processor elements and the relative time of count and communication.

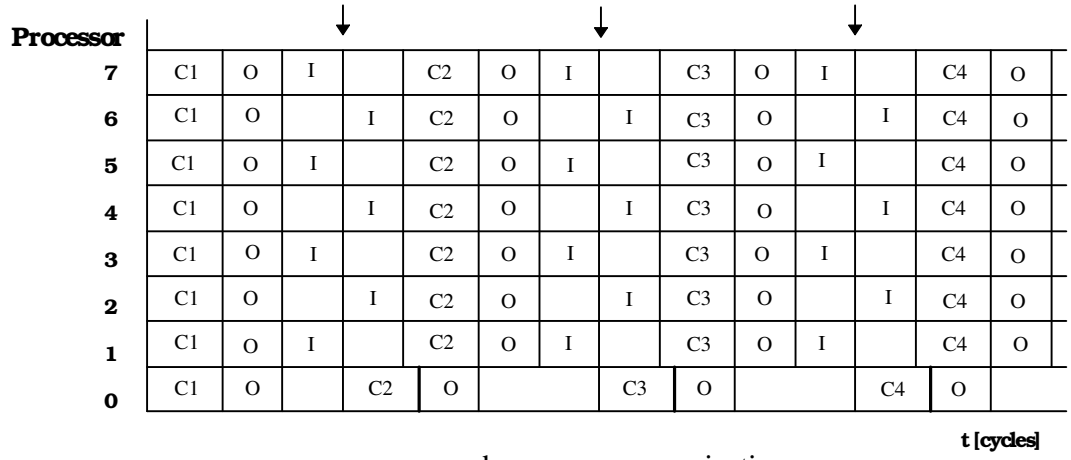
Processor activity diagram for program of convolution



a. classical pipeline



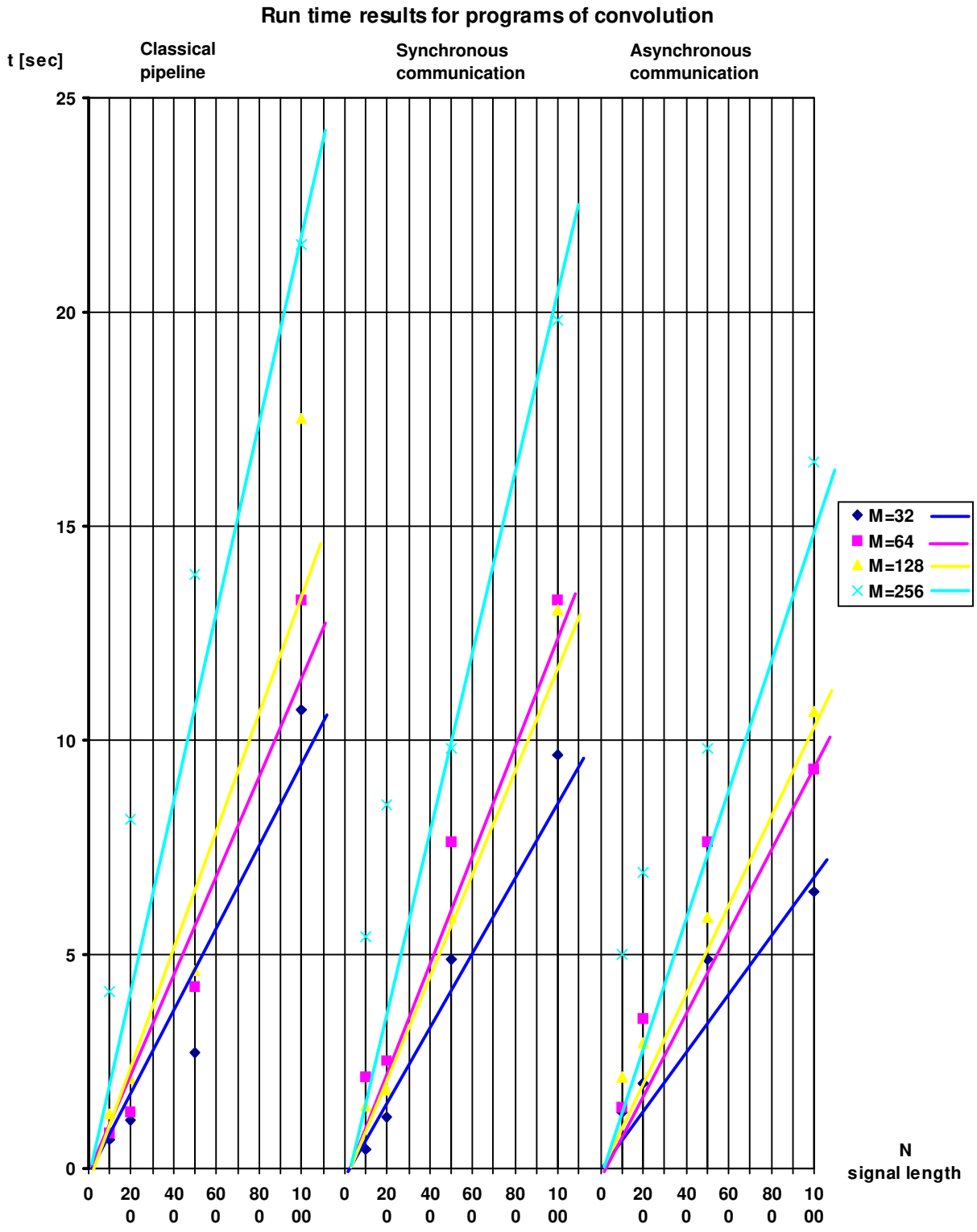
b. synchronous communication



c. asynchronous communication

Ci - count
O - output of data
I - input of data
□ - wait
↓ - begin of next cycle

The time diagram shows that the most efficient method of realization is to use the asynchronous communication between the processor elements, while he efficiency of parallel algorithm can be found by angle of time line to axis of signal lenght. It means that the synchronous communication can be also effective in the case of a relative long counting process in the processor elements.


Run time results for programs of convolution
t [sec]
Classical pipeline
Synchronous communication
Asynchronous communication
25
20
15
10
5
0
0
200
400
600
800
1000
M=32
M=64
M=128
M=256
N
signal length

3.2. MA (FIR) filter

The algorithm for the MA filter is given in formula 3.2. In point of view of the realization it is nothing else as replacing *w(i)* to b_k and *u(i)* to *x(i).*

$$y(i) = \sum_{k=0}^{q} b_k x(i - k), \qquad (3.2)$$

The list of the program is the following:

```
/*   fir_asynch.c

     Program for realization of
          FIR (MA) filter
                with
     asynchronous communication
   between the processor elements

*/

#include <stdio.h>
#include <virt_top.h>
#include <sys/comm.h>
#include <sys/time.h>

#define      BACKWARD 0      /*virtual link from previous processor*/
#define      FORWARD  1      /*virtual link to next processor*/
#define      N        1000   /*number of input signal*/
#define      Q        256    /*number of processor element*/

int main (void)
{
  int          first, second;
  double         work_time;

  int          k;
  int          x[N];
  int          b[Q];
  int          y_send = 0;
  int          y_recv = 0;
  int          id;
```

```
int         topId;
int         reqId = 0;
PipeData_t    *pipeData = NULL;

int         pipe_len = Q;
int         error;

topId = MakePipe(reqId, pipe_len, MINSLICE , MAXSLICE,
                MINSLICE, MAXSLICE, MINSLICE, MAXSLICE);
AInit(topId, -1, -1);
pipeData = GetPipe_Data(topId);

id = pipeData->id;

/* assignment for signal sequences x[i] and b[k] */
for (k = 0; k < N; ++k)
    x[k] = k;
for (k = 0; k < pipe_len; ++k)
    b[k] = k;

/* filter function */

   switch(pipeData->status){
        case PIPE_NONE:
            break;

        case PIPE_HEAD:
            for (k = 0; k < N+pipe_len-1; ++k){
                if (k > N-1)
                    y_send = 0;
                else
                    y_send = b[pipe_len-1-id] * x[k];
                ASend(topId, FORWARD, &y_send, sizeof(int), &error);
            }
            break;

        case PIPE_IN:
            for (k = 0; k < N+pipe_len-1; ++k){
                if (k > N-1)
                    y_send = y_recv + 0;
                else
                    y_send = y_recv + b[pipe_len-1-id] * x[k];
                ASend(topId, FORWARD, &y_send, sizeof(int), &error);
            ARecv(topId, BACKWARD, &y_recv, sizeof(int), &error);
```

```
                ASync(topId, BACKWARD);
            }
            break;

        case PIPE_TAIL:
            first = TimeNowHigh();
            for (k = 0; k < N+pipe_len-1; ++k){
                if (k > N-1)
                    y_send = y_recv + 0;
                else
                    y_send = y_recv + b[pipe_len-1-id] * x[k];
                ARecv(topId, BACKWARD, &y_recv, sizeof(int), &error);
                ASync(topId, BACKWARD);
                printf("%8d\n", y_send);
            }
            second = TimeNowHigh();
            work_time = (double) (second-first)/ (double) CLK_TCK_HIGH;
           printf("\nWork time: %6.24f sec\n", work_time);
           break;
    }
FreePipe(topId);
return 0;
}
```

3.3. AR (IIR) filter

The algorithm for the AR filter is given in formula 3.3.

$$y(i) = \sum_{k=1}^{p} a_k y(i-k) + x(i) \qquad (3.3)$$

The realization of the AR filter is the following:

```
/*   iir.c

    Realization of AR filter
   with synchronous communication
     between the processors

*/
#include <virt_top.h>
#include <sys/time.h>
```

```
#include <vt_lib.h>
#include <sys/root.h>
#include <sys/link.h>
#include <stdlib.h>
#include <sys/sys_rpc.h>
#include <unistd.h>

#define  N  6
#define  FORWARD  1
#define  BACKWARD 0

int topid;
int reqid = 0;

void
main(void)
{
  int pipe_len = nProcs, temp;
  PipeData_t *pipeData = NULL;
  unsigned int first, second;
  double   work_time;
  int       i, k, error, pid;
  int       x[N+1], y[N+1], *a;

  topid = MakePipe(reqid, pipe_len, 0, DimX-1, 0, DimY-1, 0, MAXSLICE);
  pipeData = GetPipe_Data(topid);

  if(pipeData->status != PIPE_NONE){

    a = (int *) malloc((pipe_len+1) * sizeof(int));
    for(i=1; i < pipe_len+1; i++) a[i] = i;

    if (pipeData->status == PIPE_TAIL){
      first = TimeNowHigh();
      for(i=0; i < N+1; ++i) x[i] = i;
      y[0] = x[0];
    }
/*************** END OF INITIALIZATION STEP ********************/

    k = 1;
    temp = 0;
    pid=pipeData->id;
    while(k <= N){
      switch(pipeData->status){
```

```
        case PIPE_HEAD:
          Recv(topid, FORWARD, &y[k-1], sizeof(int));
          Send(topid, FORWARD, &temp, sizeof(int));
          temp = y[k-1] * a[pipe_len-pid];
          break;
        case PIPE_IN:
          Recv(topid, FORWARD, &y[k-1], sizeof(int));
            Send(topid, BACKWARD, &y[k-1], sizeof(int));
            Send(topid, FORWARD, &temp, sizeof(int));
          Recv(topid, BACKWARD, &temp, sizeof(int));
            temp += y[k-1] * a[pipe_len-pid];
            break;
          case PIPE_TAIL:
            Send(topid, BACKWARD, &y[k-1], sizeof(int));
            Recv(topid, BACKWARD, &temp, sizeof(int));
          y[k] = temp + a[1] * y[k-1] + x[k];
            break;
      }
      k++;
    }
  }
  if (pipeData->status == PIPE_TAIL){
    for(i=0; i < N+1; i++) printe("%d\n", y[i]);
    second = TimeNowHigh();
    work_time = (double) (second-first)/ (double) CLK_TCK_HIGH;
    printf("\nWork time: %6.2f sec\n", work_time);
    fflush(stdout);
  }
  FreePipe(topid);
  return;
}
/*********  END OF PROGRAM ********************?
```

3.4. ARMA filter

The algorithm for the ARMA filter is given in formula 3.4.

$$y(i) = \sum_{k=0}^{q} b_k x(i-k) + \sum_{k=1}^{p} a_k y(i-k) \qquad (3.4)$$

The realization of the ARMA filter combine the topology of AR filter with algorithm of MA filter.

```
/*     arma.c

   Realization of ARMA filter
with synchronous communication
     between the processors

*/
#include <virt_top.h>
#include <sys/time.h>
#include <vt_lib.h>
#include <sys/root.h>
#include <sys/link.h>
#include <stdlib.h>
#include <sys/sys_rpc.h>
#include <unistd.h>

#define  N  6
#define  FORWARD  1
#define  BACKWARD 0

int topid;
int reqid = 0;

void
main(void)
{
   int pipe_len = nProcs, temp;
   PipeData_t *pipeData = NULL;
   unsigned int first, second;
   double   work_time;
   int       i, k, error, pid;
   int       x[N+1], y[N+1], *a, *b;

   topid = MakePipe(reqid, pipe_len, 0, DimX-1, 0, DimY-1, 0, MAXSLICE);
   pipeData = GetPipe_Data(topid);

   if(pipeData->status != PIPE_NONE){

     pid=pipeData->id;
     a = (int *) malloc((pipe_len+1) * sizeof(int));
     b = (int *) malloc(pipe_len * sizeof(int));
     for(i=1; i < pipe_len+1; i++) a[i] = 1;
     for(i=0; i < pipe_len; i++) b[i] = 1;
```

```
    for(i=0; i < N+1; ++i) x[i] = 1;

    if (pipeData->status == PIPE_TAIL){
      first = TimeNowHigh();
     y[0] = b[0]*x[0];
    }
    else {
      temp = b[pipe_len-pid-1]*x[0];
    }
/*************** END OF INITIALIZATION STEP ********************/

    k = 1;
    while(k <= N){
      switch(pipeData->status){
       case PIPE_HEAD:
        Recv(topid, FORWARD, &y[k-1], sizeof(int));
        Send(topid, FORWARD, &temp, sizeof(int));
        temp = b[pipe_len-1-pid] * x[k] + y[k-1] * a[pipe_len-pid];
        break;
       case PIPE_IN:
        Recv(topid, FORWARD, &y[k-1], sizeof(int));
          Send(topid, BACKWARD, &y[k-1], sizeof(int));
          Send(topid, FORWARD, &temp, sizeof(int));
        Recv(topid, BACKWARD, &temp, sizeof(int));
          temp += b[pipe_len-1-pid] * x[k] + y[k-1] * a[pipe_len-pid];
          break;
        case PIPE_TAIL:
          Send(topid, BACKWARD, &y[k-1], sizeof(int));
          Recv(topid, BACKWARD, &temp, sizeof(int));
        y[k] = temp + b[pipe_len-1-pid] * x[k] + a[1] * y[k-1];
          break;
     }
     k++;
    }
 }
 if (pipeData->status == PIPE_TAIL){
  for(i=0; i < N+1; i++) printe("%d\n", y[i]);
  second = TimeNowHigh();
  work_time = (double) (second-first)/ (double) CLK_TCK_HIGH;
  printf("\nWork time: %6.2f sec\n", work_time);
  fflush(stdout);
 }
 FreePipe(topid);
 return;
```

```
}
```

Bibliography

. PARIX Release 1.2 Software Dokumentation. - March 1993. - PARSYTEC Computer GmbH, Aachen, Germany.
. Cok, Ronald S.: Parallel Programs for the Transputer. - Prentice Hall. Englewood Cliffs, New Yersey 07632
. dr. Kacsuk Peter - Ferenczi Szabolcs: Parhuzamos es konkurrens programozas soktranszputeres rendszereken. - BME Mernoktovabbkepzo Intezet, Budapest,1993.
. Kernighan, Brian W. - Ritchie, Dennis M.: The C Programming Language. Second Edition. - Prentice Hall. Englewood Cliffs, New Yersey 07632