Parallel Implementations of the Simplex Algorithm

Richard Marciano, Teodor Rus

Department of Computer Science
The University of Iowa
Iowa City, IA 52242

Abstract

Three parallel implementations of the simplex algorithm on three different parallel architectures, are presented and compared. Each machine is the representative of one class of parallel computers. Performance comparisons and the major difficulties encountered by the user of these machines are given.

The potential for parallel programming of the array processors is investigated with the MPP machine. The multiprocessor systems with asynchronous shared memory are studied by implementing the simplex algorithm on the Encore machine in both the process creation by *fork()* and tasking environment. The class of supercomputers represented by the Alliant FX-8 "mini-supercomputer" where a Fortran compiler can parallelize and vectorize DO loops is considered.

Keywords: array processor, parallel programming, performance, simplex algorithm, multiprocessor, vectorization.

1 The Simplex Algorithm

The simplex algorithm was developed by Dantzig[DANT63] for finding the solution of a linear programming problem. Its simplicity and elegance made it the essential numeric tool for solving optimizing linear problems. Therefore, it was (and still is) the object of intense study [VAJD60], [FICK61], [BORG80]. Our paper is a contribution toward efficient implementations of the simplex algorithm on parallel processors available today. The general form of a linear programming problem can be expressed as follows:

Maximize the linear function $f = c_1x_1 + c_2x_2 + \ldots + c_nx_n$ *where* $c_1, c_2, \ldots c_n$ *are given real numbers called costs and* $x_1, x_2, \ldots, x_n$ *are unknowns subject to the linear restrictions*

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1i}x_i + \ldots + a_{1n}x_n &\leq b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2i}x_i + \ldots + a_{2n}x_n &\leq b_2 \\
\ldots & \quad \ldots \\
a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{ji}x_i + \ldots + a_{jn}x_n &\leq b_j \\
\ldots & \quad \ldots \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mi}x_i + \ldots + a_{mn}x_n &\leq b_m
\end{aligned}$$

and $x_i \geq 0, i = 1, 2, \ldots n$, *where* $a_{ji} \in R, i = 1, 2, \ldots, n, j = 1, 2, \ldots, m$.

The standard simplex algorithm [BUND84] consists of a sequence of iterations that for a given solution $\mathcal{X}^0 = (x_1^0, x_2^0, ..., x_n^0)$ of the linear programming problem improves f until the optimum solution is obtained (if one exists, otherwise the absence of a solution is specified). Let $\mathcal{X}^i$ be the solution before iteration i. A new solution $\mathcal{X}^{i+1}$ is constructed at iteration i from $\mathcal{X}^i$ with the property that $f(\mathcal{X}^{i+1}) \geq f(\mathcal{X}^i)$. The construction of $\mathcal{X}^{i+1}$ from $\mathcal{X}^i$ is performed by the following sequence of operations:

1. Find the variable x_c which generates the best contribution to the value of f if introduced in the solution. The index c of this variable is given by the maximum coefficient of the function f at this iteration.
2. Find the variable of $\mathcal{X}^i$ that needs to be replaced by x_c. The index of this variable is given by the smallest number $b_j/a_{jc}, j = 1, 2, \ldots, m$, for $a_{jc} > 0$. Let it be b_r/a_{rc}.
3. Transform the matrix of the initial problem by a Gaussian elimination using element a_{rc} as a pivot, i.e., perform the operation $\forall j, i, j \neq r, i \neq c, a_{ji} := a_{ji} - a_{ri}a_{jc}/a_{rc}$.

Computationally the algorithm can be presented as in figure 1.

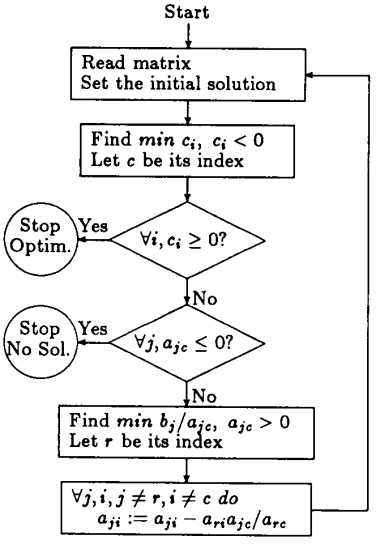

Figure 1: Flow of control

In order to use this algorithm to solve a linear programming problem the set of m linear inequalities defining the problem is first converted into a set of m linear equations by introducing at most m slack variables and by changing the sign of all free terms such that $b_j \geq 0, j = 1, 2, \ldots, m$. The optimizing function f is then added as line 0 of the linear system of equations thus obtained in the form $C - f = 0$ where C is its optimal value (originally 0), i.e., $-c_1x_1 - c_2x_2 - \ldots - c_nx_n = 0$. The initial feasible solution is then obtained by introducing m new variables whose coefficients in the function f are set to zero. The matrix of this system with the free terms in column 0 is organized as the two dimensional array called *simplex tableau*:

$$\begin{array}{cccccccc}
a_{00} & a_{01} & \ldots & a_{0i} & \ldots & a_{0n} & a_{0n+1} & \ldots & a_{0n+m} \\
a_{10} & a_{11} & \ldots & a_{1i} & \ldots & a_{1n} & a_{1n+1} & \ldots & a_{1n+m} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
a_{j0} & a_{j1} & \ldots & a_{ji} & \ldots & a_{jn} & a_{jn+1} & \ldots & a_{jn+m} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
a_{m0} & a_{m1} & \ldots & a_{mi} & \ldots & a_{mn} & a_{mn+1} & \ldots & a_{mn+m}
\end{array}$$

The simplex tableau is automatically constructed by the procedure reading the simplex matrix.

Using the simplex tableau defined above, the simplex algorithm can be formulated as the following sequence of steps:

1. Find the pivot column, i.e., perform the computation:

$$C = \begin{cases} -1, & \text{if } T[0,i] \geq 0 \text{ for } i = 1, 2, \ldots, n+m; \\ c \geq 1, & \text{if } T[0,c] = min(T[0,i]),\ T[0,i] < 0, \\ & \quad i = 1, 2, \ldots, n+m. \end{cases}$$

   If $C = -1$, the optimum solution was found. Otherwise step 2 follows.

2. Find the pivot line, i.e., perform the computation:

$$L = \begin{cases} -1, & \text{if } T[j,C] \leq 0 \text{ for } j = 1, 2, \ldots, m; \\ r \geq 1, & \text{if } T[r,C] = min(T[j,0]/T[j,C]), \\ & \quad T[j,C] > 0, j = 1, 2, \ldots, m. \end{cases}$$

   If $L = -1$ there is no solution. Otherwise step 3 follows.

3. Transform the simplex tableau by the formula:

```
for i = 0,1,...,n+m, i ≠ C do
    for j = 0,1,...,m, j ≠ L do
        T[j,i] := T[j,i] - T[L,i]*T[j,C]/T[L,C];
```

Clearly parallelism can only be found within each of these three steps. In order to obtain maximum speed, the granularity of parallelization needs to be controlled by the user according to the architecture of the machine. This is done by allowing the user to define the unit of parallelization as being a contiguous block of H lines and K columns of the simplex tableau. The number N of contiguous blocks T[H,K] (i.e., parallel jobs) in which the simplex tableau T can be partitioned is determined by:

$$p = [(n+m)/K] + sg(rest((n+m)/K))$$
$$q = [m/H] + sg(rest(m/H)), N = p * q.$$

The constants H and K that determine p, q are dependent on the type of hardware and its computation power.

2 Implementation on the MPP

The MPP machine[NASA88] is an array processor that operates under the control of a conventional VAX-11/780 front-end (figure 2) and consists of three main units:

1. Array Processing Unit, APU, a two dimensional 128 × 128 mesh with wrap around connections between processing elements in the same row or column denoted by PE(i,j), $i, j = 0, 1, \ldots, 127$. Each PE(i,j) is 1-bit processor containing 1,024 bits of random access memory denoted here by PEM(i,j)[0..1023].

2. Array Control Unit, ACU, which executes scalar operations and controls the operations performed by the APU. The ACU, figure 2, is actually composed of three units:

   - Main Control Unit, MCU, which is the local memory of the MPP used to store an MPP program and its scalar data.
   - I/O Control Unit, IOCU, which controls the flow of data in and out of the APU, in particular data transfers between the APU and STM.
   - Processing Element Control Unit, PECU, which controls the execution of the array operations in the MPP program.

3. Staging Memory, STM, which is a large storage unit of 32 megabytes connected to the APU via a fast 128 bit data path. It is used to buffer data due to limited memory capacity of the APU.

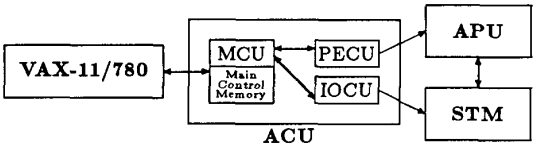

Figure 2: MPP Diagram

An MPP program is a sequential program which contains array operations, I/O operations and scalar operations. The special feature of the ACU is that all of its three control units can operate simultaneously to allow overlapping of the three types of operations found in an MPP application program.

The software support for parallel processing implemented on the MPP allows a user to develop a program using *parallel arrays* and operations on parallel arrays as computation units. A parallel array is an abstraction for the APU, i.e., an array of size 128 × 128 of a given type (integer, real, or boolean). An array operation (i.e., having parallel arrays as operands) is simultaneously performed by every PE(i,j) of the APU, each PE(i,j), $i, j = 0, 1, \ldots, 127$ acting in a lock step fashion on the corresponding memory components of the parallel array operands stored in its memory area.

The Pascal language has been extended with new constructs supporting array processing and implemented on the MPP under the name MPP Pascal.

MPP Pascal supports all the Pascal data types. In addition it has been extended with the *parallel array* as a predefined data type supporting arithmetical and logical operations and the *stager array* as a predefined data type supporting information exchange between APU and STM. Additional language constructs

operating on parallel arrays are provided in MPP Pascal allowing parallel array management in a high level fashion such as: *max, min, sum, prod, shift, rotate, transpose, rowbroad, colbroad, insert, extract.*

A parallel array may not be indexed directly. This is why to perform an array operation using selected PE-s of the APU, MPP Pascal provides the special *where* masking statement:

$$\textbf{where} < mask > \textbf{do} < S_1 > \textbf{otherwise} < S_2 >.$$

The *mask* is a boolean expression that evaluates to a parallel array of type boolean mapped on a bit-plane. Each element (i,j) in this bit-plane specifies a processor PE(i,j) of the APU enabling (mask(i,j)=1) or disabling (mask(i,j)=0) its execution.

The operations of information exchange between front-end, array memory and stager memory are shown on figure 3.

The stager memory is treated as an extended array memory of 512 parallel arrays of reals or integers. The unit of transfer is a parallel array.

The synchronization operations *waitq*, which idles the MCU until the PECU has finished, *waitio* which idles the MCU until the completion of the I/O transfer initiated by the IOCU occurs, allow the three components of the MPP to operate concurrently.

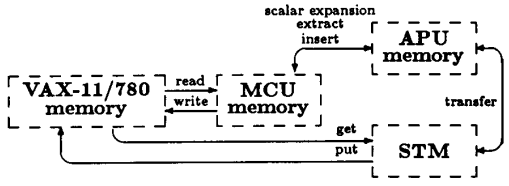

Figure 3: MPP Data flow

A typical MPP application program consists of two parts: an MPP part that runs in the MPP control unit (MPP Pascal), and a host part (Fortran, C, Pascal) that runs on the host. The MPP part is usually the "main" program consisting of data transfers and large scale array computations. The host part generates data files to be read in by the MPP main program and serves as a driver for the MPP program. The communication between MPP part and host part is performed by a tool called CAD. Each part is compiled separately on the host and resulting object modules are linked together to produce a program image that can be directly loaded into the MPP for execution. This discussion will be illustrated further with the MPP simplex implementation.

In order to execute a program the user invokes its executable image through the CAD user interface. Access to the MPP is done on a first come first served basis. Thus, the interaction between the host part and the main part of an MPP program during its execution observes a master-slave relationship.

2.1 Simplex Algorithm on MPP

The structure of array memory and stager memory determines the parallelization strategy of the simplex algorithm. It consists of splitting the simplex tableau in as many contiguous subtableaux T[128,128] as possible.

The implementation of the simplex algorithm on the MPP consists of two programs, a program running on VAX called *Prepare_data* and a program running on the MPP called *Simplex*. The program *Prepare_data* written in Fortran performs as follows.

1. Read the dimensions m, n of the simplex tableau maintained as a VAX file and determine constants p, q by the rules:

$$p = [(n+m)/128] + sg(rest((n+m)/128))$$

$$q = [m/128] + sg(rest(m/128))$$

2. Reorganize the simplex tableau T[0..m,0..n+m] as an array of parallel arrays stored on VAX file F2 in the following format:

$$\begin{array}{cccc} P(1,1) & P(1,2) & \ldots & P(1,p) \\ P(2,1) & P(2,2) & \ldots & P(2,p) \\ \ldots & \ldots & \ldots & \ldots \\ P(q,1) & P(q,2) & \ldots & P(q,p) \end{array}$$

$$P(i,j) = T[(i-1)q..(i-1)q+127, (j-1)p..(j-1)p+127].$$

3. Use CAD to invoke *Simplex*, and to wait for its execution.

The program *Simplex* is an MPP Pascal program. It uses the following type declarations:

```
type
    ParAr23  = parallel array[1..23,0..127,0..127] of real;
    ParAr1   = parallel array[0..127,0..127] of real;
    ParArInt = parallel array[0..127,0..127] of integer;
    ParArBol= parallel array[0..127,0..127] of boolean;
    StAr1    = stager array[0..127,0..127] of real;
    StAr512  = stager array[1..512,0..127,0..127] of real;

program Simplex(input,output,row_index,col_index,T1,T2);
%include 'type.dat'
%include 'procedures.dat'
var
  T1 : text; T2 : file of StAr1; A : ParAr23;
  T : StAr512; C_ind, R_ind, Pivcol, Pivrow : integer;
  K1, K2, Flag : integer; DONE, IMPOSSIBLE : boolean;
begin
 zeroarr;
 DONE := false;
 IMPOSSIBLE := false;
 Load(T1,T2,T,A,Flag);
 if (Flag <> 0) then
   repeat
     if (Flag = 23) then
       begin
         PivCol(A,C_ind,Pivcol,K2);
         if (C_ind <> -1) then
           begin
             PivRow(A,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
             if (R_ind <> -1) then
               Update(A,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
             else NoSolution := true;
           end
         else Done := true;
       end
```

```
    if (Flag = 512) then
      begin
       StPivCol(T,C_ind,Pivcol,K2);
       if (C_ind <> -1) then
         begin
          StPivRow(T,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
          if (R_ind <> -1) then
            StUpdate(T,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
          else NoSolution := true;
         end
       else Done := true;
      end
  until (Done or NoSolution)
end.
```

The *Load* procedure recomputes the constants K1 and K2 by the rules shown above and reads the file F2 into the array memory or stager memory depending upon its size. Therefore, the simplex on the MPP operates in two modes distinguished by the variable *Flag*. When $Flag \leq 23$ the entire simplex tableau is stored in the array memory and *PivCol, PivRow, Update* are then used. When $23 < Flag \leq 512$ the simplex tableau is stored in the stager memory and parallel arrays need to be *swapped-in and swapped-out* in order to be processed and updated. The procedures *StPivCol, StPivRow, StUpdate* similar to *PivCol, PivRow, Update* need to be used in that case.

Let us suppose for sake of clarity that the simplex tableau is small enough to be entirely mapped onto one parallel array $A[1,,]$. Each entry (i,j) in the tableau is thus mapped onto its own processor PE(i,j). Once pivot column and row have been determined, the tableau updating can be carried out simultaneously by all PE-s in one array operation. To perform this updating each PE(i,j) needs to access three tableau items, (i,j), (piv_row,j) and (i,piv_col). The last two tableau items are not accessible by PE(i,j) and data communication and exchange between processors PE(i,j), PE(piv_row, j) and PE(i, piv_col) is necessary. This is performed by creating two new parallel arrays (BrPC and BrPR) both constructed by using *shift* and *broadcast* array functions as carried out by the *Update* procedure. PE(i,j) now has access to the three corresponding tableau items and the tableau updating is performed by $A[1,,] := A[1,,] - BrPR \times BrPC$, figure 4. When the tableau maps over more than one parallel array in the array memory, this data broadcasting scheme is applied iteratively to each parallel array.

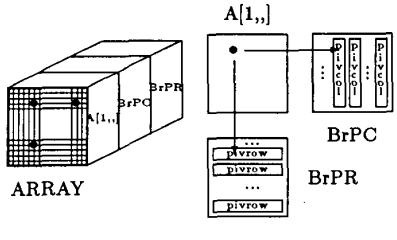

Figure 4: Simplex tableau updating operation

2.2 Performance Measurements

Performance measurements of the simplex implementation discussed above are given in the table 1. The lines of the table are labeled by the number of iterations required to find the solution while the columns are labeled by the number of parallel arrays required to store the simplex tableau. The time in seconds taken by the MPP to solve a problem of the size the number of parallel arrays recorded in the column j and performing the number of iterations recorded in the line i is recorded in the table entry (i,j).

	1	4	9	16	25	36	49	64
3	0.003	0.006	0.011	0.016	0.43	0.57	0.73	0.91
6	0.006	0.013	0.022	0.033	0.84	1.10	1.40	1.80
44	0.047	0.102	0.173	0.260	6.13	8.18	10.0	12.0

Table 1: Performance measurements on MPP

3 Implementation on Multimax

A new class of computers [GORD87] called *Multimax* emerges as multiprocessor computers using as components microprocessors that have the speed and functionality of mid-range supercomputers. The Encore Multimax is a modular system designed as a component of the *Encore Computing Continuum*[ENCO87], which provides a true multiprocessing and distributed environment. The Encore Continuum uses tightly coupled multiprocessing, distributed, and intelligent control of I/O devices and clustering of Multimax systems. A multimax cluster incorporates from 2 to 20 32-bit processors each provided with 32K byte cache of fast static RAM, 4 to 32 Mbytes of fast shared memory and configurable I/O devices.

The Multimax support for parallel program development and execution consists of a library of functions that extend the collection of system calls supported by Unix[1] and allow the user to create parallel processes in a program, schedule them for execution while sharing resources, and control their interaction. A user can take advantage of these functions creating and managing a process environment or a tasking environment.

3.1 Simplex with Process Environment

The process environment is provided by the *fork()* system call that allows a program to create processes in the user program. The function *MakeProcs* was designed by us in order to allow the simplex user to create a variable number of processes.

Process interaction is done by all processes having access at the variable declared shared. There are two classes of system calls in the parallel library allowing the user to declare shared objects:

1. When shared memory is statically managed the user proceeds as follows:
   - Declare a C-language data structure, say *data* and/or a pointer to it, say *datapt*.
   - Call the function *share()* in the parallel library to make *data, datapt* shared under the form

     datapt = share(0, sizeof(data));

[1]Unix is a trademark of Bell Labs

2. When shared memory is dynamically managed then the user proceeds as follows:

   - Provides the memory area to be manipulated dynamically by the program using the call

     alloc = share_malloc_init(size);

     which returns a pointer to an area of memory of size "size".
   - Manage dynamically the memory pointed to by alloc using calls of the form

     datapt = share_malloc(sizeof(data));
     share_free(datapt);

Process synchronization is done by using the lock data types supported by the Encore Multimax[RUS88]:

- **Lock**: is a binary semaphore supporting the operations *spin_init(lock, flag), spin_create(flag), spin_unlock(lock), spin_condlock(lock)*, where *flag* shows the state of the lock.
- **Barrier**: allows a fixed number of processes to synchronize at a given point in a program. It supports the operations *barrier_create(count, state), barrier_init(lock, count, state), barrier(lock)* where count is the number of processes that need to arrive at the barrier before it opens.
- **Semaphore**: is general semaphore supporting the operations *semaphore_init(lock, state), semaphore_create(state), semaphore_wait(lock), semaphore_signal(lock).*
- **Event**: provides a barrier at which a variable number of processes can wait having two states, *event_posted* and *event_cleared.* It supports the functions *event_create(state), event_init(lock, state), event_post(lock), event_clear(lock), event_wait(lock).*

The *lock* parameter is a pointer to an object of type the type supporting the function using it, *state* is *SPIN_BLOCK, PROCESS_BLOCK, TASK_BLOCK*, showing the mechanism implementing wait, *count* is an integer and *flag* is PAR_LOCKED or PAR_UNLOCKED.

The lock variables used in a program need to be created in shared memory. All operations supported by the lock data types specified above are atomic. In addition, the parallel library provides the function *timer_init()* and *timer_get()* which allow the timing of the program execution.

The structure of a parallel program under process environment is illustrated by the following sketch of the simplex implementation.

```
#include <stdio.h>
#include <parallel.h>
#define cols 1200
#define lines 1200
struct shared_area
  {
   double pivot, a[lines][lines+cols];
   int m, n, H, K, Procs, Jobs, JobCount;
   int C, L, p, q, ColCount, RowCount;
   BARRIER barr;
   LOCK lock;
   } *glob;
int IdProc = 0;
main (int argc, char *argv[])
   {
    int i, State = SPIN_BLOCK;
    glob = share (0, sizeof(*glob));
    /* Read matrix, parameters, and initialize data */
    spin_init(&glob- >lock, PAR_UNLOCKED);
    barrier_init(&glob- >barr, &glob- >Procs, State);
    IdProc = MakeProcs(&glob- >Procs-1);
Start: PivCol(&glob- >K,&glob- >C);
    barrier(&glob- >barr);
    if (&glob- >C < 0) { PrintSolution(); exit()}
    PivRow(&glob- >H,&glob- >L);
    if (IdProc == 0)
      &glob- >JobCount = 0;
    barrier(&glob- >barr);
    TransformL(&glob- >L, &glob- >K);
    barrier(&glob- >barr);
    if (&glob- >L < 0) { NoSolution(); exit()}
    i = Monitor(&glob- >JobCount);
    while (i < &glob- >Jobs)
          {
            Update(i/p, i%p, &glob- >H, &glob- >K);
            i = monitor(&glob- >JobCount);
          }
    barrier(&glob- >barr);
    TransformC(&glob- >C, &glob- >H);
    barrier(&glob- >barr);
    goto Start;
   }
```

The functions *PivCol(), PivRow(), Update(), TransformL(), TransformC()*, and *Monitor()* implement the three steps of the simplex algorithm, perform matrix transformations and ensure computation consistency, respectively.

3.2 Simplex with Tasking Environment

Since process creation is a very costly operation, the tasking mechanism was developed to support parallel program development on the Encore Multimax. A task is a function provided with its own stack and thus capable of being executed in parallel with other tasks. A parallel program using the tasking environment consists of a collection of tasks that can be executed in parallel. There is a special task called *master* that starts the execution of the program initiating other tasks. Each task in turn can start other tasks. The tasking environment of a program is thus defined by the memory size *Mem* used to allocate stacks for the tasks, the number of processes *Procs* that run tasks in parallel and the master task, *Master*. The tasking environment of a program and the start of the master task are set up with a call to the function *task_init*

task_init(Mem, Procs, Master, Stack, Argc, Arg0,...,Argn);

The starting of a task specified by a function *Func* in the tasking environment (by master task and/or by other tasks) is performed by the call

task_start(Func, Stack, Argc, Arg0, ..., Argm);

which allocates *Stack* bytes as the stack of this task from *Mem*,

transmits arguments on the stack and starts a process to execute the code of *Func* on this stack, if there exists a processor available for this purpose.

The tasking environment is controlled by the program using the following tasking primitives: *task_suspend()* that suspends its caller; *task_resume(name)* makes the task *name* reschedulable; *task_stop()* terminates its caller; *task_join()* waits for all tasks initiated by its caller to terminate; *task_self()* returns the task identification number of its caller.

There are two restrictions imposed on parallel program development by the tasking approach: the code of a function designed as a task needs to be provided in the program text before the invocation of that task and the locks need to be created and initialized in the main program. The consequence is a bottom-up approach for program development. The structure of a parallel C language program using the tasking environment is illustrated by the following sketch of the tasking version of the simplex algorithm.

```
#include < stdio.h >
#include < parallel.h >
#define Stacks 20000
#define Stack 500
#define cols 1200
#define lines 1200
double pivot, a[lines][lines+cols];
int m, n, Procs, JobCount, Jobs, H, K;
int answer, ColCount, RowCount, C, L, p, q;
LOCK *lock;
transform()
  {
   int i;
   i = Monitor(JobCount);
   while (i < Jobs)
        {
         Update(i/p, i % p, H, K);
         i = Monitor(JobCount);
        }
  }
master ()
   {
    int i;
    Start: ColCount = 0;
    for (i = 0, i < Procs, i++)
       task_start(Stack, PivCol, 1, C);
    task_join();
    if (C < 0) { answer = 1; return }
    RowCount = 0;
    for (i = 0, i < Procs, i++)
       task_start(Stack, PivRow, 1, L);
    task_join();
    if (L < 0) { answer = -1; return }
    JobCount = 0;
    for (i = 0; i < Procs; i++)
       task_start(Stack, transform, 0);
    task_join();
    goto Start;
   }
```

```
main (int argc, char *argv[])
   {
    /* Read matrix, parameters and initialize data */
    lock = share (0, sizeof(LOCK));
    spin_init(lock, PAR_UNLOCKED);
    Set_timers();
    task_init (Stacks, Procs, master, Stack, 0);
    Get_timers();
    if (answer == 1) PrintSolution();
    else PrintNoSolution();
   }
```

3.3 Performance Measurements

The performance measurements of the simplex implementation on the Multimax using the process environment and the tasking environment closely follow the same pattern. Therefore in tables 2 and 3 we only present the performance of the program implemented in the process environment which is slightly better than for the tasking environment. Table 2 illustrates the variation of the time to solve a problem whose simplex tableau was 512×512 (i.e., $256 \times K$ elements), with the number of processes running in parallel and the granularity of their interaction. The lines of this table are labeled by the number of processes running in parallel and the columns are labeled by the granularity of the process interaction. The granularity is expressed by the size of the contiguous subtableaux of the simplex tableau transformed by a process independently of the other processes. This is given in the number of parts in which the lines and columns of the simplex tableau are divided. The time in seconds needed by the Encore parallel processor to solve the problem is recorded in the entry (i,j). However, examining the behavior of the algorithm on a large number of problems we observed that the best time was provided by job size (16, 260) with 12 processors. Therefore, the last column of table 2 records the behavior of the algorithm for this process interaction granularity.

	1	2	4	8	16	32	(16,260)
1	33.6	9	20	27.2	32.4	35.3	17.8
3	33.9	9.4	7.2	9.8	11.1	11.7	6.6
6	34.4	9.7	5.3	5.8	6.3	6.5	4.2
9	34.8	10.2	3.6	4.6	5	5.2	3.6
12	35.2	10.7	4.1	4.5	4.6	4.7	3.5
15	35.7	11.1	4.7	4.5	4.5	4.6	3.9
18	39.9	12.8	6	4.9	5.3	5.3	4

Table 2: Granularity study

	1	4	9	16	25	36	49	64
3	0.6	2	2.5	3.5	4.6	5.5	6.7	8.2
6	1	2.1	3.5	5	6.6	8.5	10.5	12.9
44	2.9	7.1	13.2	21.9	36.4	44.3	62.5	73.2

Table 3: Encore performance measurements

4 Implementation on Alliant

The Alliant FX8 is a register to register machine equipped with 8 MC68000 compatible vector processors, 11 interactive processors for input and output, and a 64 megabyte memory subsystem [ARGO86]. Parallel processing on the Alliant is performed by

pipelining vector operations and by parallel processing of the 8 vector processors.

Parallel programming support is provided by the Concentrix operating system (Unix-based) and the FX/Fortran language which supports the array data type. Like MPP Pascal, FX/Fortran has been extended with a set of intrinsic array functions : *min(Ar), max(Ar), size(Ar), etc...*, as well as a very similar conditional array assignment statement allowing masking on an array assignment:

where < *cond* > < *st*1 > **otherwise** < *st*2 > **end where**

A Fortran optimizing compiler generates parallel streams of control and vector operations unfolding DO loop operations under programmer control.

Four modes of program execution are available on the Alliant FX8, concurrent execution, vector execution, vector-concurrent execution, and concurrent-outer vector-inner execution. Typically a programmer tunes the program execution by inserting compiler directives in the Fortran code. These directives might for example either turn off vectorization for a specific loop or rather force concurrency specifying that there are no data dependencies in a loop. The information allowing the programmer to inject compiler directives in his program is provided by the compiler itself. There are only 7 groups of possible directives. The syntax of a compiler directive is **CVD$**[*s*] *directive* where:

$$s = \begin{cases} G, & \textit{directive} \text{ applies globally (i.e., to the end of file);} \\ R, & \textit{directive} \text{ applies to end of the current routine;} \\ L, & \textit{directive} \text{ applies to end of the current loop (default).} \end{cases}$$

The available *directives* are (* indicates the default value): *ASSOC, NOASSOC**, telling the compiler to perform the optimization of the associative operations; *CNCALL, NOCNCALL**, allowing subroutine and function references in loops optimized for parallel execution; *CONCUR*, NOCONCUR*, forcing (or inhibiting) the optimization for concurrency irrespective of data dependency; *DEPCHK*, NODEPCHK*, telling the compiler to check (or to inhibit the checking) for data dependencies between loop iterations; *LSTVAL*, NOLASTVAL*, telling the compiler to generate code to save last values of original indexes and promoted scalars of optimized loops and arrays; *SYNC*, NOSYNC*, telling the compiler to check for synchronization problems between loop iterations; *VECTOR*, NOVECTOR*, telling the compiler to optimize (or to inhibit the optimization) for vectorization.

4.1 Simplex on Alliant FX8

In the Alliant FX8 simplex implementation through compiler directives, we inhibited vectorization of certain loops and strategically forced concurrency by inhibiting data dependency checks. A sketch of the Fortran version of the simplex algorithm fully tuned and optimized and running on all eight processors follows:

```
      program simplex
      integer L, C, m, n
      real mat(1200, 1200)

CVD$R NOLSTVAL
C Read matrix and initialize data
      SetTimer
99
C Find Pivot col and find Pivot row
C Divide Pivot row
CVD$L NOSYNC
      do 80 j = 1, m
         if(j.ne.L) then
CVD$L NOSYNC
            do 40 k = 1, n+m
               if(k .ne. C) then
                mat(j,k)=mat(j,k)-mat(L,k)*mat(j,C)
               endif
40          continue
            endif
80       continue
C Divide Pivot col
         goto 99
C GetTimer
C Print solution or lack of solution
         stop
         end
```

In this implementation all the code is brought in the main program to avoid subroutine calls.

4.2 Performance Measurements

In order to compare the performance of the simplex algorithm implemented on the three machines, MPP, Encore Multimax, and Alliant FX8, we run the program on the same set of problems and organize the results in the same way. The behavior of the algorithm on the Alliant FX8 is given in table 4.

	1	4	9	16	25	36	49	64
3	0.015	0.093	0.39	0.79	1.3	1.8	2.5	3
6	0.026	0.15	0.74	1.6	2.5	3.6	4.9	5.9
44	0.11	0.8	4.6	9.5	16.7	24.5	33.6	41.2

Table 4: Alliant performance measurements

5 Instead of Conclusion

The conclusions of the experiments we performed with the simplex algorithm implemented on the three different computers are twofold. On the one hand they regard the efficiency of the algorithms implemented on the new parallel processing architectures measured by the speed-up obtained by their parallelization and on the other hand they regard the user convenience of the various parallel processing architectures measured by the difficulties implied in their programming.

The speed-up of *Machine1* versus *Machine2*, (*Machine 1, Machine 2* are A for Alliant, E for Encore, and M for MPP) while solving a problem requiring a given number of iterations for various sizes of the simplex tableau is recoded in a line of a speed-up tableau labeled by *Machine1:Machine2* in table 5. The size of the simplex tableau used in our experiments is measured in number of parallel arrays required to accommodate it.

	1	4	9	16	25	36	49	64
3 - *A:E*	40	20.4	7.7	5.6	4.4	3.9	3.5	3.5
3 - *M:A*	5.2	14.8	36.1	48.2	3	3.2	3.4	3.3
3 - *M:E*	207	302	278	268	13.3	12.3	12	11.6
6 - *A:E*	38.5	14	4.7	3.1	2.6	2.4	2.1	2.2
6 - *M:A*	4.3	11.6	33.6	48.5	3	3.3	3.5	3.3
6 - *M:E*	167	162	159	152	7.9	7.7	7.5	7.2
44 - *A:E*	26.4	8.9	2.9	2.3	2.2	1.8	1.9	1.8
44 - *M:A*	2.3	7.8	26.6	36.5	2.7	3	3.4	3.4
44 - *M:E*	61.7	69.6	76.3	84.2	5.9	5.4	6.2	6.1

Table 5: Speed-up for 3, 6 and 44 iterations

The simplex problems for these sizes were actually automatically generated from smaller problems. Therefore, instead of definitive conclusions we present our findings as the following three observations:

- The simplex algorithm provides a natural application in which operations on matrices are used. Therefore, array and vector processors should perform better than the general multiprocessor machines. This was confirmed by the speed-up of the algorithm implemented on the three machines.
- The second conclusion shows that even for problems that are naturally suited for array and vector operations, the control of the granularity of process interaction allows the shared memory multiprocessor to become comparable in efficiency to the vector processor in the case of large size problems.
- The third conclusion shows that the performance of the vector processor provided with parallel execution becomes comparable with that of the array processor when the the size of the problem is large. This is due to the cost of array transfer between array memory and stager memory.

Parallel processors clearly allow the simplex algorithm to become an efficient tool in solving linear programming problems. Therefore, comparing the standard version of the simplex algorithm [TARJ83],[DANT79] with the newly discovered polynomial time algorithms [ASPV79], [BORG80](pp. 18-22) may provide different data when executed in parallel environments. So, further study of the parallel implementations of the simplex algorithm and its comparison with the parallel implementations of these newer methods are necessary.

Each of the three different philosophies of handling parallel processing has its specific type of user difficulties. The major difficulties in programming an *array processor* result from the promotion of the array (which is a defined type in most programming languages) to a predefined data type. However, the predefined type "array" does not coincide with the array type existing in most languages nor with the matrix type existing in mathematics. Therefore, in order to take advantage of the machine's potential for parallel processing both experience and the language support developed in this respect provide the necessary help. The major difficulties in developing parallel programs for a *multiprocessor machine* result from the requirement to explicitly manage the implicit process type in the program. This task is performed by the multiprogramming (multiprocessing) operating system operating on a sequential program. Again, experience, the development of concepts and their encapsulation in appropriate data types in the language seem to provide the real help. As for developing parallel programs through the *compiler* the major difficulties result from the compiler-programmer-processor interaction which requires the programmer to have knowledge of architecture, compilers, and the behavior of the algorithm. Therefore, this could be only a temporary solution used to successfully parallelize existing code that would otherwise be too expensive to redesign.

6 Acknowledgments

We would like to express our acknowledgments to Prof. G. Carmichael who provided us with the possibility to learn and use the MPP machine, to Prof. E. Haug and D. Golden, for allowing us to use the Alliant FX8 and Encore Multimax in the HSCF of the University of Iowa, Iowa City, and to Daniela Rus, for the valuable observations and suggestions leading to improvements of our work.

References

[ARGO86] Argonne National Laboratory, *Using the Alliant FX/8*, ANL/MCS-TM-69, Rev. 1, Mathematics and Computer Science Division, September 1986.

[ARGO87] Argonne National Laboratory, *Using the Encore Multimax*, ANL/MCS-TM-65, Rev. 1, Mathematics and Computer Science Division, February 1987.

[ASPV79] Aspval, B., Stone, R. E., "Khachian's linear programming algorithm", J. Algorithms, 1 (1980) pp 1-13

[BORG80] Borgwardt, K. H., *The Simplex Method*, A Probabilistic Analysis, Springer-Verlag, New York 1988.

[BUND84] Bunday, B. D., *Basic Linear Programming*, Edward Arnold, London 1984.

[DANT63] Dantzig, G. B., *Linear Programming and Extensions*, Princeton Univ Press, Princeton, NJ 1963

[DANT79] Dantzig, G. B., "Comments on Khachian's Algorithm for linear programming", Tech. Report SOR 79-22, Dept. Operations Research, Stanford Univ, Stanford, CA, 1979

[ENCO87] Encore Multimax, *Using the Encore Multimax* Argonne National Laboratory, MCS-TM-65, pp. 1-1, 1987.

[FICK61] Ficken, F. A., *The Simplex Method of Linear Programming*, Holt, Rinehart and Winston, New York, 1961.

[GORD87] Gordon, Bell, "The Multi - A New Computer Class", *Using The Encore Multimax*, Argonne National Laboratory, MCS-TM-65, pp. 5-8, 1987.

[NASA88] NASA Goddard Space Flight Center, Greenbelt, Maryland, *MPP Pascal Programmer's Guide*, March 1988.

[RUS88] Rus, T., "Language Support for Parallel Programming", Proceedings of Computer Standards Conference, pp. 21-23, March 21-23, Washington, D.C., 1988.

[TARJ83] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics Philadelphia, Pennsylvania 1983.

[VAJD60] Vajda, S., *Linear Programming and the Theory of Game*, John Wiley and Sons, New York, 1960.