INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Development of Parallel BLAS with ARCH Object-Oriented Parallel Library, Implementation on CM-5

J.M. Adamo* †

TR-94-045

September 1994

Abstract

This paper reports on the developmemt of BLAS classes using the *ARCH* library. The BLAS library consists in two new *SpreadMatrix* and *SpreadVector* classes that are simply derived from the ARCH *SpreadArray* class. Their implementation essentially makes use of the ARCH *remote read and write* functions together with barrier-synchronization. They provide a good illustration of how ARCH can contribute to the development of loosely-synchronous systems. This paper describes the architecture of SpreadMatrix and SpreadVector classes and illustrates their use through the construction of a neural-network simulator.

[1]International Computer Science Institute, Berkeley, USA

[2]Université Claude_Bernard, Lyon, France

1 Introduction.

The *ARCH* library is a general purpose object-oriented library that we have recently written to support the development of portable parallel applications. A description of ARCH can be found in [1]. ARCH is intended to offer the right set of specific programming facilities that are needed to develop parallel applications, no matter what the type of involved algorithms could be: synchronous, loosely synchronous, or asynchronous. The library has been written in C++. The initial implementation is for the Thinking Machine's CM5, although care has been taken to ensure maximum portability. The library allows parallel program development in the SPMD style. It has been designed so the C++ compiler can perform a complete type checking of programs written with it. The library consists of a layered set of programming facilities that are pictured in figure 1. The most basic objects in the library are threads. A *thread* can execute in an independent stack and be subject to the application of operators performing: creation, initialisation, scheduling, suspension. Threads can be used to construct more disciplined objects: processes. A *process* is a thread that can be handled within a special function *par* which automatically performs process creation, initialization and synchronized termination. The library provides a set of *synchronous message-passing* and *remote read/write* functions. The first ones allow point-to-point synchronous communication between asynchronous processes while the second can be used in read-only shared memory programs or combined with barrier-synchronization to provide the right tool needed for loosely synchronous system programming. The last layer of the library deals with spread arrays and pointers. A *spread array* provides a global shared virtual memory that can be accessed via indexation from any processor and *spread pointers* generalize regular pointers for transparent data access in spread array. The new *SpreadMatrix* and *SpreadVector* classes that are described in the present report are simply derived from *SpreadArray*s (figure 1). Their implementation essentially makes use of the ARCH remote read and write functions together with barrier-synchronization. They provide a good illustration of how ARCH can contribute to the development of loosely-synchronous systems. This paper describes the architecture of *SpreadMatrix* and *SpreadVector* classes and illustrates their use through the construction of a neural-network simulator. The presentation is completed by a set of appendices that give a clear idea of the whole design. Other works dealing with the development of Object-oriented BLAS libraries can be found in[9] and [10].

2 Developing Object-oriented parallel BLAS with ARCH.

2.1 Spread matrices and vectors.

2.1.1 SpreadMatrix Definition.

Spread matrices are defined by their type and the geometry of the set of processors which the matrices are spread over. The geometry naturally is a grid whose dimensions have to be specified by the user. For instance, in the neural-net simulator that we have developed with the library (see Appendices 3-5), the processor grid is specified as a 4*8-grid as we developed the simulator on a 32-node CM5. The dimensions could instantly be changed by simply changing the template instantiation parameters if the simulator were to run on different machines. The processors are numbered in row major order. This numbering actually is a

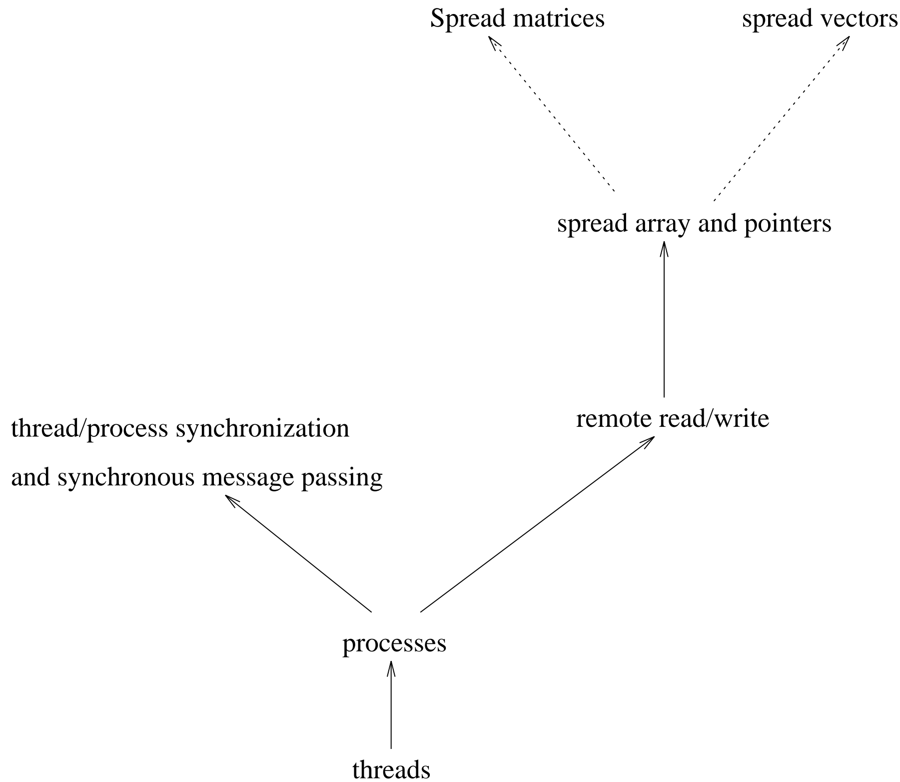

Figure 1: Architecture of ARCH library

logical one. The mapping from logical to physical numbers is given by the *mapping()* function that is supposed to be defined by the user. ARCH provides the predefined *mapping()* function identity in its *environment* file.

The spread matrix facility is architectured as shown in Figure 2. $SpreadMatrices < T, VPROCSIZE, HPROCSIZE >$ is the base class that inherits from the $SpreadArray < T >$ class and provides most of functions found in the current implementation. Derived from this class are two other classes: SpreadMatrix and DSpreadMatrix. The SpreadMatrices, SpreadMatrix and DSpreadMatrix types are to be respectively used to get: fixed size matrices defined dynamically, fixed size matrices defined statically, variable size matrices defined dynamically.

The spread matrix set of classes is shown in Appendix 1. The lines below define a table of pointers to three different spread matrices. The first one is a "dynamic" spread matrix whose dimensions are initialized to 512 and 1024. These dimensions can be changed dynamically. The next two ones are "static" spread matrices. Further illustration can be found in the neural-network simulator code (see Appendices 3-5).

$$DSpreadMatrix < T, 512, 1024, 4, 8 > SM0;$$
$$SpreadMatrix < T, 128, 512, 4, 8 > SM1;$$
$$SpreadMatrix < T, 512, 256, 4, 8 > SM2;$$
$$SpreadMatrices < T, 4, 8 > *SM[3];$$

```
template <class T, int VPROSIZE, int HPROCSIZE>
 SpreadMatrices : SpreadArray<T>{};

    template <class T, int VPROSIZE, int HPROCSIZE>
     DSpreadMatrix : SpreadMatrices<T, VPROCSIZE, HPROCSIZE>   {};

    template <class T, int VPROSIZE, int HPROCSIZE>
     SpreadMatrix : SpreadMatrices<T, VSIZE, HSIZE, VPROCSIZE, HPROCSIZE>   {};
```

Figure 2: SpreadMatrix class Architecture

2.1.2 Spread matrices as spread arrays.

A *SpreadMatrices* actually is a *SpreadArray* as defined in the ARCH library (see [1]). In the current version, I choose the simplest spreading policy that consists in block-decomposing the matrices over the processor grid so matrix-block i is held by processor i (logical numbering). This is also the way the matrices are decomposed in the TMC's native CMSSL libraries accessible via C* and CM_FORTRAN. A SpreadMatrix of dimension Vs*Vh spread over a VPROCSIZE*HPROCSIZE processor grid actually is encoded as a SpreadArray with three spread dimensions with size: Vs/VPROCSIZE, VPROCSIZE, HPROCSIZE, and one internal dimension with size: Hs/HPROCSIZE. Depending on architectures, other spreading policies would, of course, be possible (and would lead to different implementations). For instance, the block-scattering policy as defined in [6] can be implemented as shown in [8]. All this actually is hidden to the final user who is provided with suitable global/local access and input/output functions (see appendix 1-2).

Figure 3 shows a 8*9-SpreadMatrix spread over a 2*3-processor grid. The SpreadMatrix is encoded as a SpreadArray with three spread dimensions of size: 4, 2, 3 and one internal dimension of size: 3. The sequences in the grid represent SpreadArray indexes.

2.1.3 Access functions.

Two access types are provided: global and local. Giving global coordinates (v, h), the global function returns the global reference, over the nodes, to the corresponding SpreadArray element. A global reference consists of a processor number (logical numbering) and a pointer to a T-type data item in the processor local memory (see [1], spread-arrays and pointers section).

```
template < class T, int VSIZE, int HSIZE, int VPROCSIZE, int HPROCSIZE >
Lval <S, T >
SpreadMatrix < T, VSIZE, HSIZE, VPROCSIZE, HPROCSIZE >
::operator()(int v, int h) {
  return SpreadArray < T >::operator()
            (v%(VSIZE/VPROCSIZE), v/(VSIZE/VPROCSIZE),
            h/(HSIZE/HPROCSIZE), h%(HSIZE/HPROCSIZE));
}
```

Giving local coordinates (v, h), the local function simply returns the pointer to a SpreadMatrix local block element.

```
template < class T, int VSIZE, int HSIZE, int VPROCSIZE, int HPROCSIZE >
T &
SpreadMatrix < T, VSIZE, HSIZE, VPROCSIZE, HPROCSIZE >
::local(int v, int h){
     return ((T *)local_ptr)[v*(HSIZE/HPROCSIZE) + h];
}
```

P0 P1 P2

000 0	000 1	000 2	001 0	001 1	001 2	002 0	002 1	002 2
100 0	100 1	100 2	101 0	101 1	101 2	102 0	102 1	102 2
200 0	200 1	200 2	201 0	201 1	201 2	202 0	202 1	202 2
300 0	300 1	300 2	301 0	301 1	301 2	302 0	302 1	302 2
010 0	010 1	010 2	011 0	011 1	011 2	012 0	012 1	012 2
110 0	110 1	110 2	111 0	111 1	111 2	112 0	112 1	112 2
210 0	210 1	210 2	211 0	211 1	211 2	212 0	212 1	212 2
310 0	310 1	310 2	311 0	311 1	311 2	312 0	312 1	312 2

P3 P4 P5

Figure 3: SpreadMatrix as a SpreadArray

2.1.4 Matrix operators.

The functions in the SpreadMatrix class are those we usually find in BLAS libraries (see Appendices 1-2). The SpreadMatrices class contains most functions: matrix multiplication,

arithmetic operations (addition, subtraction, element-wise multiplication), matrix-to-vector horizontal and vertical reduction, vector-to-matrix horizontal and vertical expansion, cartesian norm, outer product. Both derived classes inherit from SpreadMatrices. The DSpreadMatrix class has two special functions that allow for dynamic expansion and shrinking of dynamic spread matrices: sm.*on(Vs, Hs)* performs a redefinition of the sm spread matrix with the new shape defined by the arguments while sm.*off()* invalidates the current shape of sm, making it unavailable down to the next redefinition. Finally, two additional functions: *sequential_to_parallel()* and *parallel_to_sequential()* are specific to the CM5 implementation and perform spread matrix switching from sequential to parallel memory and vice-versa.

2.1.5 Spread vectors.

A SpreadVectors facility with an architecture similar ot that of SpreadMatrices is also available. The SpreadVectors architecture and functions are presented in Appendix 2.

3 Implementation of the BLAS.

3.1 Implementation notes.

The matrix-product functions operate in the parallel space while the element-wise functions can operate both on sequential and parallel memory. For instance, in the parallel neural-net simulator, that is briefly described later on, and in Appendices 3-5, the data are loaded initially in the parallel space by the class constructor and next kept and operated on in this space. In other applications, the data could possibly not be loaded once and for all in the parallel space, so it would not be worth performing the element-wise operations in the parallel space. Indeed, reading and writing data to/from parallel memory would be too expensive as compared to the benefit we could expect by performing the functions there. The current implementation has been written using the Nodal TMC's BLAS from the CMSSL library [12] and a few very low level functions that can be found in the CM5 CMRTS run-time library [13]. I could have instead used the AC compiler [4] which generates code that can be executed on the vector units. I preferred using CMSSL as I expected to get optimized code from there.

4 Writing a neural-network simulator.

4.1 Architecture.

The BLAS classes described aboved have been used to develop a neural-net simulator (see Appendices 3-5). The simulator can execute the learning back-propagation algorithm for nets of any dimensions (number of layers: second template parameter) and at two possible precision levels (float or double: first template parameter). The network can currently perform both the batch and the on-line back-propagation learning algorithms. The user is required to provide the table of layer and pattern dimensions (table N[], Appendix 3), the keep_on_learning condition and the data-base, together with the functions required to load it on the nodes of machine (see Appendix 4). The code of the batch back-propagation algorithm is diplayed in Appendix 5. This code is a very high level one that could be

executed on any other sequential or parallel machine as well (provided, of course, we have the related implementation of the SpreadMatrices and Vectors libraries). A version of the simulator is currently available on the Berkeley's CM5.

5 Performance issues and Concluding notes.

The library has been assessed. Timing one instance of the multiplication function yielded times only 10 percent slower than those I got with the same function from the CMSSL library. The current implementation is the first draft I wrote and can probably be improved. The element-wise operations make direct use of the CMRTS functions and should have the same efficiency as the same operations performed within C* or CM-Fortran. There is a source of inefficiency left in the current construction of the neural-network simulator: the TANH() function is not currently executed on the vector units. The reason is that the arithmetic functions provided by CMRTS are only accessible at the assembly level and I thought unreasonable spending time to do that. I wonder why TMC does not make these functions available at the node level and, more generally, why a clean interface for using the vector engine at this level is not provided? Trying to use the vector units in the current status of system development is some kind of nightmare. Further details on the performances achieved by the neural-network simulator can be found in the companion paper [2].

Acknowledgments. I would like to thank Davide Anguita (University of Genova and ICSI) for his friendly collaboration. We shared the work of writing the neural-network simulator and he performed its assessement.

References

[1] J.M. Adamo. *Object-Oriented Parallel Programming: Library design and development for SPMD Programming.* Research-report, ICSI TR-94-011, February 1994.

[2] J.M. Adamo, D. Anguita. *Object-Oriented Design of a BP Neural Network Simulator and Implementation on the Connection Machine CM5.* Research-report, to appear, 1994.

[3] E. Anderson et al. *LAPACK, Users' guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[4] W.W. Carlson, J.D. Schlesinger. *AC: A C Language and compiler for the CM-5 Node Architecture.* SSuperComputing Researc Center, SRC-TR-93-094, 1993.

[5] J. Choi et al. *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers.* Oak Ridge National Laboratory, 1992.

[6] J. Choi et al. *PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers.* Oak Ridge National Laboratory, May, 1993.

[7] J. Choi et al. *PB-BLAS Reference Manual.* Oak Ridge National Laboratory, March, 1994.

[8] D. Culler et al. *Parallel Programming in Split-C.* Technical Report U.C. Berkeley, 1993.

[9] J.J. Dongara et al. *LAPACK++: A Design and Overview of Object-Oriented Extensions for High Performance Linear Algebra.* Oak Ridge National Laboratory, 1992.

[10] J.J. Dongara et al. *An Object oriented Design for High Performance Linear Algebra on Distributed Memory Architectures.* Oak Ridge National Laboratory, 1992.

[11] S.L. Johnsson, L.F. Ortiz. *Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages for array syntax.* TMC, TR-226, 1992.

[12] Thinking Machine Corporation. *CMSSL for CM Fortran: CM5 Edition.* Volume I, II, January 1993.

[13] W.R. Swanson, K. Crouch. *Connection Machine Run Time System (CMRTS), Architectural Specification, Version 7.2.* TMC, March, 1993.

[14] Thinking Machine Corporation. *CM-5, VU Programmer's Handbook.* Volume I, II, August 1993.

Appendix 1.

```
template<class T, int VPROCSIZE, int HPROCSIZE>
class SpreadMatrices: public SpreadArray1<T>{

   public:

      int Vsize;
      int Hsize;

      //the two following declarations are  needed for
      //vectorized computation on CM5

      int location;  //sequential: 0, parallel: 1

      CMRT_desc_t descriptor[HPROCSIZE/VPROCSIZE];

      SpreadMatrices(int Vs, int Hs)
        :SpreadArray1<T>(Vs/VPROCSIZE, VPROCSIZE, HPROCSIZE, Hs/HPROCSIZE, 3)
      {
         if( HPROCSIZE < VPROCSIZE || HPROCSIZE%VPROCSIZE
            ||  Vs%VPROCSIZE  || Hs%HPROCSIZE || Vs%HPROCSIZE){
            SelectPrintf("on node %d: incorrect matrix size in constructor\n",
                           self_address);
            preempt();
```

```
      }

       Vsize = Vs;
       Hsize = Hs;

       location = 0;
    }

    ~SpreadMatrices(){}

    Lval1<S, T> operator()(int v, int h){
      if(location){
          SelectPrintf("on node %d: access to matrix element in parallel
                        space not currently implemented\n",  self_address);
          preempt();
      }
      return SpreadArray1<T>::operator()(v%(Vsize/VPROCSIZE),
                                          v/(Vsize/VPROCSIZE),
                                         h/(Hsize/HPROCSIZE),
                                          h%(Hsize/HPROCSIZE));
      //a Lval is returned only when location == 0
    }

    T& local(int v, int h){
      if(location){
          SelectPrintf("on node %d:  access to matrix element in parallel
space not currently implemented\n", self_address);
          preempt();
      }
      return ((T *)local_ptr)[v*(Hsize/HPROCSIZE) + h];
      //a T& is returned only when location == 0
    }

    void sequential_to_parallel_space();
              //switching from sequential space to vector space on CM5

    void parallel_to_sequential_space();
              //switching from vector space to sequential space on CM5

    void C_equal_A_mult_B
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
    void C_equal_A_mult_tB
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
```

```
void C_equal_tA_mult_B
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_tA_mult_tB
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_A_mult_B
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_A_mult_tB
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_tA_mult_B
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_tA_mult_tB
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_A_plus_B
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_A_plus_v_B
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_A_minus_B
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_A_minus_v_B
                  (T u, T v,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_A_ewmult_B
                   (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                    SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_A
                  (T u,
```

```
                       SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A);
      void C_equal_u_plus_A
                      (T u,
                       SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A);
      void C_equal_u_minus_A
                      (T u,
                       SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A);
      void Vector_to_Matrix_Vexpansion
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V);
      void Matrix_to_Vector_Vreduction
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V);
      void columnwise_reduction(void *buff_in, void *buff_out, T);
      void rowwise_scattering_V
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V,
                                  void *buff_out , T);
      void Vector_to_Matrix_Hexpansion
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V);
      void Matrix_to_Vector_Hreduction
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V);
      void rowwise_reduction(void *buff_in, void *buff_out, T);
      void rowwise_scattering_H
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *V,
                       void *buff_out, T);
      void C_equal_A_outer_B
                      (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                       SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_u_D_plus_v_A_outer_B
                      (T u, T v,
                       SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *D,
                       SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                       SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      T glob_norm();
};

template<class T, int VSIZE, int HSIZE, int VPROCSIZE, int HPROCSIZE>
class SpreadMatrix: public SpreadMatrices<T, VPROCSIZE, HPROCSIZE>{

   public:

      SpreadMatrix()
       : SpreadMatrices<T, VPROCSIZE, HPROCSIZE>(VSIZE, HSIZE)
      {}

      ~SpreadMatrix(){}
```

```
};

template<class T, int VPROCSIZE, int HPROCSIZE>
class DSpreadMatrix: public SpreadMatrices<T, VPROCSIZE, HPROCSIZE>{

   public:

      DSpreadMatrix(int Vs, int Hs)
       : SpreadMatrices<T, VPROCSIZE, HPROCSIZE>(Vs, Hs)
      {}

      ~DSpreadMatrix(){}

      // redefine with a new shape the matrices it is applied to
      void on (int Vs, int Hs);

      // undefine the current shape down to a new redefinition
      void off()

};
```

Appendix 2.

```
template<class T, int VPROCSIZE, int HPROCSIZE>
class SpreadVectors: public SpreadArray1<T>{

   public:

        int size;

        //descriptor is needed for vectorized computation on CM5
        //sequential: 0, parallel: non-zero pointer value
        CMRT_desc_t descriptor;

   public:

      SpreadVectors(int s)
         :SpreadArray1<T>(HPROCSIZE*VPROCSIZE, s/HPROCSIZE*VPROCSIZE, 1)
      {
         if( HPROCSIZE < VPROCSIZE || HPROCSIZE%VPROCSIZE
            || size%(VPROCSIZE*HPROCSIZE)){
            SelectPrintf("on node %d: incorrect vector size in constructor\n",
                           self_address);
            preempt();
         }
         size = s;
```

```
   descriptor =0;
}

~SpreadVectors(){}

Lval1<S, T> operator()(unsigned i){
  if(descriptor){
      SelectPrintf("on node %d: access to vector element in parallel
                    space not currently implemented\n", self_address);
      preempt();
  }
  return SpreadArray1<T>::operator()(i/(size/(VPROCSIZE*HPROCSIZE)),
                                        i%(size/(VPROCSIZE*HPROCSIZE)));
  //a Lval is returned only when descriptor == 0
}

T& local(int i){
  if(descriptor){
      SelectPrintf("on node %d: access to vector element in parallel
                    space not currently implemented\n", self_address);
      preempt();
  }
  return ((T *)local_ptr)[i];
  //a T& is returned only when descriptor == 0

}

void sequential_to_parallel_space();
void parallel_to_sequential_space();
void C_equal_A_mult_B
                  (SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_A_mult_B
                  (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_A_mult_B
                  (T u, T v,
                   SpreadVectors<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
void C_equal_u_D_plus_v_A_mult_B
                  (T u, T v,
                   SpreadVectors<T, VPROCSIZE, HPROCSIZE> *D,
                   SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                   SpreadMatrices<T, VPROCSIZE, HPROCSIZE> *B);
```

```
      void C_equal_A_plus_B
                       (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_u_A_plus_v_B
                       (T u, T v,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_A_minus_B
                       (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_u_A_minus_v_B
                       (T u, T v,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_A_ewmult_B
                       (SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *B);
      void C_equal_u_A
                       ( T u,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A);
      void C_equal_u_plus_A
                       (T u,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A);
      void C_equal_u_minus_A
                       (T u,
                        SpreadVectors<T, VPROCSIZE, HPROCSIZE> *A);
      T glob_norm();
};

template<class T, int SIZE, int VPROCSIZE, int HPROCSIZE>
class SpreadVector: public SpreadVectors<T, VPROCSIZE, HPROCSIZE>{

   public:

      SpreadVector()
        : SpreadVectors<T, VPROCSIZE, HPROCSIZE>(SIZE)
      {}

      ~SpreadVector(){}

};

template<class T, int VPROCSIZE, int HPROCSIZE>
class DSpreadVector: public SpreadVectors<T, VPROCSIZE, HPROCSIZE>{
```

```
   public:

      DSpreadVector(int s)
        : SpreadVectors<T, VPROCSIZE, HPROCSIZE>(s)
      {}

      ~DSpreadVector(){}

      void on (int s);

      void off();
}
```

Appendix 3.

```
#include "environment"
#include "blas-incl"

class Root_proc: public Process{
   public:
      Root_proc()
                :Process((membfunc_ptr)&Root_proc::body){}

      void body(){

         //first three: layer dimensions, last: # of patterns
         int N[3+1] = {128, 256, 128, 1024};

         //neural_net instantiation
         Neural_net<float, 3>neural_net(&N[0], "input.data", "target.data");

 //learning
         neural_net.back_prop(&N[0]);
      }
};

MAIN(root_proc, Root_proc)
```

Appendix 4.

```
template<class T, int L>
class Neural_net{
//T = type of data (float or double)
//L =  \#levels

//forward
```

```
SpreadMatrices<T, 4, 8> *S[L];

SpreadMatrices<T, 4, 8> *W[L-1];

SpreadVectors<T, 4, 8> *B[L-1];

//backward
SpreadMatrices<T, 4, 8> *D[L-1];

SpreadMatrices<T, 4, 8> *DW[2][L-1];

SpreadVectors<T, 4, 8> *DB[2][L-1];

SpreadMatrices<T, 4, 8> *TG_ptr;

DSpreadMatrix<T, 4, 8> *aux_ptr;

int SWITCH;
T eta;
T alpha;
T e_new;
T e_old;

//const
T c_alpha;
T c1_eta;
T c2_eta;

   public:

      //constructor
      Neural_net(int *N, char* in_file, char *t_file);

      //learning
      void back_prop(int *N);

      //keep_on_learning condition (user defined function)
      int keep_on_learning(int iter);

      //matrix and vector initialization (user defined functions)
      void data_file(SpreadMatrices<T,4,8> *Matrix, char *data_file);

      void matrix_initialization(SpreadMatrices<T,4,8> *Matrix, T Range);

      void vector_initialization(SpreadVectors<T,4,8> *Vector, T Range);
```

```
};
```

Appendix 5.

```
// backprop parameters
#define ALPHA        0.9
#define ETA          0.75
#define ACCELERATION 1.05
#define DECELERATION 0.7
#define SATURATION   2.94
#define RESIZE       0.9

template<class T, int L>
Neural_net<T, L>::Neural_net(int *N, char* in_file, char *t_file){

      for(int i=0; i<L; i++){
         S[i] = new SpreadMatrices<T, 4, 8>(N[L], N[i]);
         if(!i) data_file(S[i], in_file);
         S[i]->sequential_to_parallel_space();
      }

      for(i=0; i<L-1; i++){

         W[i] =   new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
         matrix_initialization(W[i], (T) N[i]+1);
         W[i]->sequential_to_parallel_space();

         B[i] =  new SpreadVectors<T, 4, 8>(N[i+1]);
         vector_initialization(B[i], (T) N[i]+1);
         B[i]->sequential_to_parallel_space();

         D[i] =  new SpreadMatrices<T, 4, 8>(N[L], N[i+1]);
         D[i]->sequential_to_parallel_space();

         DW[0][i] =  new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
         DW[0][i] =  new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
         DW[0][i]->sequential_to_parallel_space();

         DW[1][i] = new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
         DW[1][i]->sequential_to_parallel_space();

         DB[0][i] = new SpreadVectors<T, 4, 8>(N[i+1]);
         DB[0][i]->sequential_to_parallel_space();

         DB[1][i] = new SpreadVectors<T, 4, 8>(N[i+1]);
```

```
         DB[1][i]->sequential_to_parallel_space();
      }

         TG_ptr =   new SpreadMatrices<T, 4, 8>(N[L], N[L-1]);
         data_file(TG_ptr, t_file);
         TG_ptr->sequential_to_parallel_space();

         aux_ptr = new DSpreadMatrix<T, 4, 8>(N[L], N[L-1]);
         aux_ptr->sequential_to_parallel_space();

         set_down_SDS_heap();

         SWITCH = 0;
         c_alpha = ALPHA;
         c1_eta  = ACCELERATION;
         c2_eta  = DECELERATION;
         eta = ETA;
}


template<class T, int L>
void Neural_net<T, L>::back_prop(int *N){

   int iter = 0;
   e_old = (T) 1.0e6;

   barrier_synchronization();

   while (keep_on_learning(iter)){
      iter++;
      int i;
      for(i=1; i<L; i++){
         S[i]->Vector_to_Matrix_Vexpansion(B[i-1]);
         S[i]->C_equal_u_D_plus_v_A_mult_B(1, 1, S[i], S[i-1], W[i-1]);
         S[i]->TANH();
      }

      D[L-2]->C_equal_A_minus_B(TG_ptr, S[L-1]);
      e_new = (T)1.0/(N[L]*N[L-1])*(D[L-2]->glob_norm());
      if(e_new < e_old){
         eta *= c1_eta;
         alpha = c_alpha;
         S[L-1]->C_equal_A_ewmult_B(S[L-1], S[L-1]);
         S[L-1]->C_equal_u_minus_A(1, S[L-1]);
         D[L-2]->C_equal_A_ewmult_B(D[L-2], S[L-1]);
```

```
        D[L-2]->C_equal_u_A((T)2.0/(N[L]*N[L-1]), D[L-2]);

        //back
        for(i=L-2; i>0; i--){
           D[i-1]->C_equal_A_mult_tB(D[i], W[i]);
           aux_ptr->on(N[L], N[i]);
           aux_ptr->C_equal_A_ewmult_B(S[i], S[i]);
           aux_ptr->C_equal_u_minus_A(1, aux_ptr);
           D[i-1]->C_equal_A_ewmult_B(D[i-1], aux_ptr);
           aux_ptr->off();
        }

        int SWITCH_new = (SWITCH+1)%2;
        for(i=0; i<L-1; i++){
           DW[SWITCH_new][i]->C_equal_u_D_plus_v_tA_mult_B
                                     (alpha, eta, DW[SWITCH][i],S[i],D[i]);
           D[i]->Matrix_to_Vector_Vreduction(DB[SWITCH_new][i]);
           DB[SWITCH_new][i]->C_equal_u_A_plus_v_B
                                (alpha, eta, DB[SWITCH][i], DB[SWITCH_new][i]);
           W[i]->C_equal_A_plus_B(W[i], DW[SWITCH_new][i]);
           B[i]->C_equal_A_plus_B(B[i], DB[SWITCH_new][i]);
       }
        SWITCH = SWITCH_new;

     }else{
           eta *= c2_eta;
           alpha = 0;
           for(i=0; i<L-1; i++){
              W[i]->C_equal_A_minus_B(W[i], DW[SWITCH][i]);
              B[i]->C_equal_A_minus_B(B[i], DB[SWITCH][i]);
           }
      e_old = e_new;
     }
  }
}
```