Res. Lett. Inf. Math. Sci., 2004, Vol.**6**, pp135-142
Available online at http://iims.massey.ac.nz/research/letters/

Parallel containers: A tool for applying parallel computing applications on clusters

M.Gan-El & K.A. Hawick†

Institute of Information & Mathematical Sciences
Massey University at Albany, Auckland, New Zealand.

† k.a.hawick@massey.ac.nz

Parallel and cluster computing remain somewhat difficult to apply quickly for many applications domains. Recent developments in computer libraries such as the Standard Template Library of the C++ language and the Message Passing Package associated with the Python Language provide a way to implement very high level parallel containers in support of application programming. A parallel container is an implementation of a data structure such as a list, or vector, or set, that has associated with it the necessary methods and state knowledge to distribute the contents of the structure across the memory of a parallel computer or a computer cluster. A key idea is that of the parallel iterator which allows a single high level statement written by the applications programmer to invoke a parallel operation across the entire data structure's contents while avoiding the need for knowledge of how the distribution is actually carried out. This transparency approach means that optimised parallel algorithms can be separated from the applications domain code, maximising reuse of the parallel computing infrastructure and libraries. This paper describes our initial experiments with C++ parallel containers.

1 Introduction

The Single Program Multiple Program (SPMD) model of parallel programming is now widespread and is well supported by message passing library implementations such as MPI[1]. MPI has bindings for the C programming language and is hence compatible at a non-object oriented level with C++[3]. There are some recent attempts to make an MPI that is more object compatible with C++. These are not yet mature enough for us to use in the work reported in this paper, but we anticipate progress in the near future.

We have experimented with use of a simple regular rectilinear storage container such as a multi-dimensional array on a SPMD environment such as a compute cluster supporting Linux and having MPI libraries and the GNU[4] compilation system.

The Standard template Library (STL)[5] associated with C++ has matured considerably over recent years and building from the excellent implementation and notes originating from SGI Inc[6], the GNU distribution has a complete STL system available for serial programs. A particularly useful starting point in the STL is a container known as a **valarray** which provides most of the conventional elastic array facilities in an object-oriented (OO) packaged library. Like the rest of the STL, valarray is implemented using the generics or template s mechanism and it is therefore possible to write serial programs like that discussed in section 2.

The idea of an Object-Oriented container that can somehow encapsulate the parallelism is not a new one[7]. We believe however that it is only recently that it has become feasible to develop a practical implementation using and interface close to that of the STL with MPI.

In section 3 we describe our implementation of a parallel valarray and the interface specification we have developed. In section 5 we discuss our conclusions and some ideas for further work to develop other parallel iterators and containers.

2 Serial valarray

The STL contains a valarray class that can be used as follows.

```
#include <iostream>
#include <valarray>
using namespace std;

int main(){

  const int N = 10;
  valarray<double> x(N), y(N), z(N);

  for(int i=0;i<N;i++ ) x[i] = (double) i;

  y = 10.0;

  z = sqrt( x + y );

  for(int i=0;i<N;i++) cout << z[i] << " ";
  cout << endl;

  return 0;
}
```

Objects x, y and z declared to be of type "valarray of doubles" and with conformant sizes are treated as generalised vectors to which we can assign individually or collectively and to which collective operations such as sqrt can be applied. The output of the code fragment is:

```
3.16228 3.31662 3.4641 3.60555 3.74166 3.87298 4 4.12311 4.24264 4.3589
```

The valarray container has a range of useful and intuitive operators that have been overloaded or supplanted to make sense for full valarrays or indeed subslices. The subslicing notation is powerful as it allows various logical tilings or data distributions[8] to be modelled. Data shifting operations such as cshift [9] also allow data distributions to be offset in a manner that is commonly used for finite difference methods in numerically solving partial differential equations and other regular geometry problems. Indeed, the whole valarray class makes use of many of the data parallel concepts that are commonly implemented in data parallel Fortran languages[10, 11]. Some of these ideas were also partially implemented in the powerful C-Star[12] data parallel C language available on the Connection Machine[13].

3 Parallel valarray_p

In this section we describe our interface specification for a parallel container. A key goal for our implementation is to achieve an interface that is intuitively as simple to use and is as close as possible in style to the existing STL container library.

The class **valarray_p**

```
template<class T>
class valarray_p {
};
```

is our attempt to create a parallel container that is similar in many aspects and also conceptually to the serial **valarray**. Users of this container benefit from parallelism without the difficulty of detailed message passing implementation and debugging. However, our valarray_p class is not yet optimised for performance. The MPI calls we have used in our prototype are mainly blocking calls. The LAM MPI[2] header we use is <mpi.h> so the MPI calls are in C and not the new version of C++ MPI calls (header file <mpi++.h>).

Since we assemble valarray_p objects dynamically we created a class

```
template<class T>
class Special_type {
  public:
    static inline MPI_Datatype datatype();
};
```

that performs the generic type mechanism for MPI through specialisation. The basic predefined MPI datatypes must all be provided. For example:

```
template<>
class Special_type<double> {
  public:
    static inline MPI_Datatype datatype() {return MPI_DOUBLE;}
};
```

The user has to supply the specialisation for any user-defined type by constructing an MPI derived datatype. For the class

```
class Structure {
  public:
    Structure();
    ~Structure();
    Structure& operator=( const Structure& obj );
    int    i;
    char   c[10];
    double d[5];
};
```

the following definition can be provided by the user

```
template<>
class Special_type<Structure> {
  public:
    static inline MPI_Datatype datatype() {
      MPI_Datatype Struct_type;
      MPI_Datatype type[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE};
      int          blocklen[3] = {1, 10, 5};
      MPI_Aint     disp[3] = {0, sizeof(int), 2*sizeof(double)};
      MPI_Type_struct( 3, blocklen, disp, type, &Struct_type);
      MPI_Type_commit( &Struct_type );
      return Struct_type;
    }
};
```

Since MPI types give a description of memory layout one should consider alignment restrictions of the hardware/compiler. In the above example we used sizeof()to find information on datatypes extent and alignment rules, but the following functions provided by MPI can be used instead:

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
int MPI_Type_size(MPI_Datatype datatype, MPI_Aint *size)
int MPI_Address(void* location, MPI_Aint *address)
```

Distributing the data can be done in four main ways. The first distribution option is to keep the data in the root (or master) processor.

The data will be distributed and gathered for every operation on the class. This has a high communication penalty for fast operations like addition, subtraction and even for square roots.

The effect of parallelisation starts to show a benefit when we apply an equation to each element of the class such as (assuming valarray_p<double>)

```
double equation(double x) {
return log( pow( exp( sqrt( (x*20.0+3.5)/2.08436 ) ), 0.0432) );
}
```

This has sufficient compute to communications ratio that some speedup is actually measurable.

The second distribution option is to have full copy of the data in the root processor as before but at the construction stage every processor will get a different section of the data.

If the data can not be distributed evenly between the processors there are two particular sub-schemes that can be chosen. One sub-scheme is distributing the data-remainder evenly between the processors. A second sub-scheme is to leave the data-remainder in the root processor's memory and distribute even portions to the other processors.

The third distribution option is to distribute a full copy of the array and assign an index scope to each of the nodes. The fourth distribution option is used if we have memory size restriction. In this case we break the data into segments and distribute between the processors and which are the only in-memory copies of data.

The principle idea behind the valarray_p class is to hide the MPI calls from the user. One code element that must happen once for all MPI calls is the communication setup. Constructors can take this role including the function MPI_Init(&argc, &argv) . The MPI_Init will be matched with MPI_Finalize() call placed in the destructor. This way the MPI is completely hidden from the user. Some issues that are covered by this are:

1. passing the arguments of main(int argc, char* argv[]) to the constructor
2. The default constructor (mainly needed to create an array of valarray_p) should be matched with a function that will get the arguments from main and setup the communication for MPI
3. The setup must be called for the entire class once which means that probably the best way is using static member function for the communication setup.

We have chosen to setup MPI in main() for simplicity. This can simply be one of the conditions for the valarray_p usage - ie a feature!

```
int main(int argc, char *argv[]) {
  MPI_Init(&argc,&argv);

  //using the valarray_p class . . .

  MPI_Finalize();
}
```

4 valarray_p Interface

In this section we describe the facilities and interface for <valarray_p>:
The class defines a one-dimensional array of type T

```
template<class T>
class valarray_p {
. . .
};
```

We define the following Constructors:

valarray_p(); The default constructor creates an empty valarray_p. This constructor is supplied to facilitate construction of arrays of valarray_p, and is expected to be followed by the resize() member function to allocate the right size.

explicit valarray_p(unsigned n); Creates a valarray_p of length n. Each element initialised to T& value = T().

valarray_p(const T& value, unsigned n); Creates a valarray_p of length n. Each element initialised by value. Note: the value comes before the number of elements. This is done to make valarray_p compatible with valarray.

valarray_p(const T* array, unsigned n); Creates valarray_p of length n. Each element initialised by the corresponding array values. Note: if n is greater than the size of array the behaviour is undefined.

valarray_p(const vector<T>& obj); Creates valarray_p of length obj.size(). Each element initialised by the corresponding obj values.

valarray_p(const valarray<T>& obj); Creates valarray_p of length obj.size(). Each element initialised by the corresponding obj values.

valarray_p(const valarray_p<T>& obj); The Copy constructor.

We also supply: **~valarray_p();** which is the destructor, and destroys all elements and deallocates memory. The constructors distribute the valarray_p data evenly between the processors leaving the data remainder at the root processor along with a full copy of the valarray_p. It is intended to develop the class to give the user the ability to choose between the distributions introduced earlier. At the construction time each object is assigned with the MPI communication size as well as the rank of the processor that will handle it. If the number of processors is greater than the valarray_p::size() the behaviour is undefined. At the moment if the number of processors is less than 2 the experimental program will crash.

We supply functions:

```
void resize( unsigned n );
void resize( unsigned n ,const T& value );
```

These resize functions are expensive and intended to be used to initialise an array of valarray_p (see default constructor). If the size grows, the additional elements are initialised by their default constructor T() or by value respectively.

In the experimental implementation the resize function does not handle memory well. It could be improved by using the idea of capacity as introduced in the STL vector. Our implementation attempts to follow the STL valarray specification.

We implement Assignment operators:

```
valarray_p& operator=( const valarray_p& obj );
```

This assigns the elements of the valarray_p obj. It checks for self assignment but if obj has different size, the behaviour is undefined.

```
valarray_p& operator=( const T& value );
```

Assigns value to each elements of the valarray_p. The assignment operator must be provided for the T type of the elements.

The assignment operator does not have a communications over-head since every processor assigns the object with the portion that it holds. However in our implementation, the root processor has to execute assignment to the full copy of the array.

The subscript operator:

```
T operator[]( unsigned index ) const;
```

Returns the value at index. The caller must ensure that index is valid, else the result is undefined.

```
void subscript( unsigned index, T& value );
```

Modifies the element at index to be value. This function replaces the function **T& operator[](unsigned index);** that returns a reference to the element. By returning a reference to the element we allow it to be modified but in parallel computing the reference has no meaning.

These functions require a redesign to find a better solution. Moreover if we distribute a full copy of the array to all of the processors the second function could be replaced with the original subscript that returns a reference.

The member functions supplied are:

```
unsigned size() const;
```

Returns the number of elements.

```
void print() const;
```

Prints the arrays elements to the screen. We overloaded the output operator so it is compatible with the C++ std::cout.

```
T max() const;
T min() const;
```

These functions return the maximum and minimum value respectively. The operators $<$ and $>$ must be provided for the T type.

```
T sum() const;
```

Return the sum of all elements. The operator += must be provided for the T type.

```
valarray_p cshift( int n );
```

Returns a new valarray_p with the elements shifted cyclically by n positions, leaving the original valarray_p unchanged. The direction of the shift depends on the sign of n If n is positive, the shift is to the left and if n is negative, the shift is to the right.

```
valarray_p apply( T (*func)(T) ) const;
```

Returns a new valarray_p where the function func was applied to each element, leaving the original valarray_p unchanged.

```
valarray_p& operator+=( const valarray_p<T>& obj );
valarray_p& operator+=( const T& value );
```

Both functions call the operator += for each element of *this with the corresponding elements of obj or value respectively. If the size of *this and obj differ, the result is undefined. The operator += must be provided for the T type. The same implementation idea could be applied to the operators -=, *=, /=, %=.

Non-member functions supplied are:

```
template <class T> valarray_p<T>
   operator+( const valarray_p<T>& obj, const T& value );
template <class T> valarray_p<T>
   operator+( const T& value, const valarray_p<T>& obj );
```

Returns a new valarray_p that contains the result of adding value to each element of obj.

```
template <class T> valarray_p<T>
   operator+( const valarray_p<T>& obj1,
              const valarray_p<T>& obj2 );
```

Returns a new valarray_p that contains the result of adding each element of obj1 to its corresponding element in obj2.

The operator += must be provided since the non-member function uses the overloaded member function operator +=. The same implementation idea could be applied to the operators -, *, /, %.

5 Summary

We plan further work to look at scalability and speedup of each member function and perhaps to calibrate an optimal number of processors for a specific action on a valarray_p class. The test work to date has focused on use of doubles (ie 64 bit floating point arithmetic). We plan to tackle more complicated user-defined structures and look at implementation where pointers are part of the user-defined structures Unfortunately at present MPI does not support these. We also plan to try to create the auxiliary classes (slice_array, gslice_array, mask_array, indirect_array) making use of the valarray slice mechanism and nomenclature. We also hope to investigate the use of parallel iterators as an underpinning item of parallel infrastructure for our containers. Parallel random access iterators would provide a useful approach to constructing non rectilinear containers.

Finally, we expect to be able to enhance the use of MPI calls by using collective communication, like MPI_Scatterv, and MPI_Gatherv, and non-blocking calls.

Acknowledgements

This work has benefited from funding from Massey University's Institute of Information and Mathematical Sciences and from use of the Helix cluster supercomputer of the Allan Wilson Centre for Molecular Ecology and Evolution and operated by Massey University. Thanks to M.Johnson for technical assistance in setting up MPI on Helix.

References

[1] The Message Passing Interface Standard, See *www.mpi-forum.org.*

[2] The LAM Message Passing Interface Implementation, See *www.lam-mpi.org*

[3] The C++ Programming Language, Bjarne Stoustrup, Addison-Wesley, ISBN 0-201-88954-4, First edition 1985.

[4] The GNU Project and Free Software Foundation, See *www.gnu.org*

[5] The Standard Template Library, see eg The Complete C++ Reference, Fourth Edition, Herbert Schildt, McGraw-Hill 2003, ISBN 0-07-222680-3.

[6] Silicon Graphics Inc. Standard tenplate Implementation, See *www.sgi.comtechstl*

[7] Massively Parallel Python, See *pympi.sourceforge.net*

[8] K-Tiling: A Structure to Support Regular Ordering and Mapping of Image Data" Oscar Bosman, Peter Fletcher and Kenneth Tsui, Proc. Australian Pattern Recognition Society Workshop on Two and Three Dimensional Spatial Data: Representation and Standards, December 1992, Perth, Western Australia.

[9] DAP Series - Parallel Data Transforms, P.M.Flanders, Active Memory Technology (formerly ICL DAP Division), 1988.

[10] The High Performance Fortran Forum. High Performance Fortran Language Specification, v 1.1. Rice University, Houston Texas http://www.crpc.rice.edu/HPFF/home.html, November 1994.

[11] S Ranka, H W Yau, K A Hawick, and G C Fox. High Performance Fortran for SPMD Programming: An Applications Overview. NPAC SCCS Report, http://www.npac.syr.edu/hpfa/Papers/HPFforSPMD/, November 1996.

[12] The C-Star, Data Parallel Programming language Manual, Thinking Machines, 1990.

[13] The Connection Machine, Fortran Programming Manuals, Thinking Machines Corporation, 1988.