# MAPS: An Efficient Parallel Language for Scientific Computing

Roy Williams
Steve Karmesin

*California Institute of Technology*

## 1. Project Summary

We propose to construct MAPS, a collection-oriented SPMD programming environment for writing parallel programs that will simplify the development and porting of many scientific algorithms, both structured and unstructured. This very general class of applications includes finite-difference, finite-element and finite-volume methods, multiblock and multigrid algorithms, PIC codes for simulating plasma physics, quadtree and octree algorithms, graph algorithms such as circuit simulation, sparse matrix algorithms, and many others.

The project is inspired and driven by:

- Execution speed. The whole point of using parallel computers is for speed; this cannot be sacrificed.
- A strong emphasis in this project will be methods of reusing existing C and Fortran code to save rewriting.
- Speed of learning. If the package is too hard to learn then it won't be used.
- Separation of algorithm from parallelism. The system can deal with load-balancing and parallelism entirely, or the user may implement custom load-balancing methods.
- Speed of porting. A MAPS code and the library itself will port easily to new, different and better machines.

All of these are essential. The proposed package will deliver tools that will improve the state of the art in writing, porting and learning without sacrificing speed of execution.

There are just two basic concepts upon which the system is based: Set and Map from elementary mathematics. Just as mathematics works by beginning with simple concepts and extending them, we extend and combine Sets and Maps in an object oriented language to build powerful and efficient tools for mathematical and scientific programming. Power comes from basing it on proven extendable concepts, and efficiency from operating on many set elements at a time and from the flexibility to use the proper implementation for particular special cases.

MAPS will be implemented as a C++ class library with the philosophy that a naive user need not know very much, yet a sophisticated user may extend the language arbitrarily. It will use inheritance and polymorphism to hide the details of how an array, for example, is a type of Map, but allow a sophisticated user to produce a new type of array with the properties needed for a specific application or computer.

A crucial part of a conventional language is that the average user need not understand code generation or the hundreds of other things involved with turning a high level language into an executable program. Here it is not simply desirable that the average user need not know just how it is built, it is part of the point of the entire exercise. The division between people who specialize in writing numerical algorithms and the people who specialize in computer architecture is real and important and on this level specialization has worked well. On the level of a production code run by a wide group of users, specialization has also been successful. In between however is a very broad band in which specialization has been less successful: writing the code itself. By basing MAPS on the concepts of "set" and "map", and by doing it in an extendable, object oriented language, we will be able to create an environment in which

it is far easier for physicists, numerical analysts, and assembly language tweekers to work together without having to become experts in each other's fields.

Because one form of Set is a direct enumeration of its elements, and because it is natural to define operations on Sets, great flexibility is available to applications in the form of dynamic set creation, a feature not usually found in collection oriented languages. This allows the use of the package to include things that are not normally considered part of the "computation": the construction and manipulation of unstructured meshes such as tetrahedral meshes, octrees, electric circuits, and other general graphs. This will remove the barrier of complexity of implementation from the unstructured methods, making them more attractive for use on a wide variety of problems.

Data is stored in long vectors to allow use of vector/pipeline operations for maximum efficiency, and sorting of sets is available to minimize cache-miss inefficiency in indirectly-addressed loops. We shall implement a lazy evaluation scheme to reduce the overheads of excessive memory allocation and excessive communication between processors.

The MAPS language is a base upon which general and flexible libraries may be added to the core language, in application areas such as load-balancing; structured, multiblock and unstructured mesh computations; elliptic and hyperbolic solvers for such meshes; solution-adaptive meshes; multigrid meshes and solvers; quadtrees and octrees; graphics. Higher levels of abstraction would be representations of continuous fields and linear operators on them. Because the language itself is based on general and flexible modules, these libraries will have a much wider range of applicability than most current ones.

With the data hiding and library generation capabilities provided by C++ and the MAPS library, it will be much easier for teams of programmers to generate large applications with some assurance that they are correct[2][3]. This is the primary reason that software developers for non-scientific codes are migrating to object-oriented languages like C++: they allow faster and more reliable production of complex codes. The future of scientific codes is in larger, more complex codes that simulate more realistic physics on varied architectures. These techniques, or ones very much like them, will be important in achieving these improvements.

# 2. Project Motivation

The first question to be asked and answered when considering a project like this is: why bother? Can the tools now available and now under construction accomplish the tasks that we want to see done, and if not, why not. We describe here why the current crop of computer languages -- though powerful and important -- are inadequate for our purposes and the reasons that they don't stand up.

## 2.1. Conventional Languages

Current programming languages and tools commonly in use for scientific codes are designed to handle one piece of data at a time and bootstrap up to handling multiple pieces by pasting loop constructs on top of single-piece-of-data instructions. The problem is, these loop constructs have meanings far beyond those we want for most many-piece-of-data operations. They imply ordering of one instruction before another, they specifically reference temporary loop variables and automating the removal of these constructs to compile for a parallel machine has proven very difficult. The key, we believe, to solving this problem is to remember that computers do what you say, not what you mean, so when you say things that you do not mean, it is very difficult for a compiler to undo the damage. This has proved particularly hard in for parallel machines.

Languages like FORTRAN have an "array" construct that represents a block of memory and a notation that lets you look up a given element. When the array is declared you say how many indices it needs (its dimension), and what range each must be in. Each use of the array must specify the correct number of indices, so a routine that is written for a 2D array must be called with a 2D array as argument or havoc will result. When writing a simulation of a two dimensional system that dimensionality -- and sometimes even the array size -- gets imprinted deep in the code. What

should be general purpose routines that perform operations that do not intrinsically care about the dimensionality of the array -- from simply zeroing all elements to an iterative linear system solver -- will have to be rewritten from the ground up with such specifics as the dimensionality of the mesh, the presence or absence of guard cells, and even the assumption of uniform memory striding built in. This wastes time as this essentially old code is rewritten, bugged and debugged time and again.

## 2.2. Collection-oriented Languages

There are now many languages in various stages of development which handle arrays of data more effectively, such as High-Performance Fortran[6], C*[14], APL[5], Paralation Lisp[1], C**[9], pC++[10], LPAR[8]. The code "`a=3*b`" can now be interpreted as multiplying each member of the array `b` by 3 and setting it to the corresponding element of the array `a`. These languages are a great help in simplifying the writing, understanding, or debugging of software, as well as allowing the compiler and run-time system more flexibility for optimization. Furthermore the compiler can execute the code on a parallel machine. For applications that need no more than flat arrays and simple transformations of them (like finite difference operators) these languages work well.

## 2.3. Arrays are Not Sufficient

A great many of the applications of today and more of tomorrow however require more than simple flat arrays. There are problems with shock waves, chemical reactions, boundary layers, gravitational collapse and many others that wildly varying time and length scales in different parts of the domain. These can make excellent use of data structures more complex than simple arrays. Other problems, such as protein folding, DNA sequencing, or grid generation virtually demand such more complex data structures.

In Fortran, a one-dimensional array is a set of data objects, labelled by integers from 1 to N. It is an essentially static structure, since adding a new object to the set requires finding a new, larger piece of memory to copy the entire array, whereas deleting an object requires copying all the objects whose subscript is greater than that of the deleted object.

If the array is distributed among the processors of a multicomputer, this inflexibility is magnified, since addition or deletion of objects in the array requires communication with all the other processors so that they can update their records of which (global) subscripts correspond to which data objects.

The array can be coerced into representing almost any data structure; indeed many Fortran hackers take this as an article of faith. Examples include graphs, meshes, stacks, lists and trees. The result is often indirect addressing, for example:

```
      do 1 icell = 1, ncell
         do 1 i = 1, 3
1           x(i,icell) = y(i,k(icell))
```

where the array `k` is an index into the array `y`. When such an application is ported to a parallel machine, some problems might be:

- Efficiency: many modern processors have a hierarchical memory consisting of registers, cache, main memory, and perhaps other levels. There is a sequence of address spaces, where the next in the sequence is larger and slower. The result of this is that when a particular memory location has been accessed, access of locations close to that location is much faster than arbitrary accesses, so that the line of code shown above executes much faster if the consecutive values of `k` are close together. It is thus desirable to sort the arrays to get close to maximum speed. The sorting code should be written only once, rather than for each application.
- Distribution: Since the memory of the parallel processor is distributed, the arrays will also be distributed. A given processor will have a subset of the `y` values, and if the index values all fall within this subset then processing can occur locally, or else the memory of another processor must be interrogated to get the `y` value.

- Local and global name-spaces: If an array of 10000 numbers is distributed uniformly among 100 processors, then a given processor may have subscripts 6701 through 6800. But these are global names, and at some software level they must be translated to subscripts 1 through 100, which is the actual amount of data the processor controls. This translation may involve wasted computer cycles. More significantly, codes with indirect addressing can be very difficult to read, with thickets of do-loops and subscripts: the addition of a further layer (local and global subscripts) makes matters even worse.
- Input and output: A file containing the values of an array should be ordered in the natural sequential order of the array, and it is difficult to avoid a sequential bottleneck when writing or reading such a file. If each processor has the same number of the array elements, it can seek to the correct place in file by knowing only its processor number, but if this is not the case, a bottleneck occurs as the processors communicate to discover where their part of the data should reside.

Each of these issues can be addressed within the context of a particular code -- the data may be calculated, sorted, distributed, located, and printed with conventional techniques and this is done every day today. The problem is that the code that performs each of those tasks must know details about all of the others, and they get woven together to the point where a conceptually simple change very often requires extensive changes to disparate sections of the code. This freezes codes into the particular architectures where they are efficient, the particular algorithms that are initially chosen, and the particular people who have invested a great deal of time writing and debugging it. The point is not that these types of problems cannot be done at all in a language with no more complex data structures than arrays; it can be done on a Turing machine too. The point is that it can be made much easier.

In the example above, the fact that `x` has two subscripts, the first of which is an integer between `1`and `ncell`, and the second is an integer between `1`and `3` is explicitly used in a section of code that is really only concerned with a permutation using the indirection array `k`. Far more intuitive is code of the form:

```
x = y(k)
```

where `x`, `y`, `k`, `y(k)` and the equal sign have meanings defined more precisely below.

If the layouts are static, the problems above can be solved by a preprocessing stage[7], where the distribution of data to processors is decided in advance. Global/local subscripts can be properly resolved, indices and arrays sorted for efficient operation, and data cached on some processors to prevent excessive communication. Unfortunately such manipulation of global arrays must be carried out on a sequential machine and communicated to the parallel machine, creating a bottleneck, and whenever the layout changes the analysis must be redone.

The problems above are magnified if the application is to change not just the values in the data arrays, but also the values in the indices, and the sizes of these arrays. Such “topological” changes might occur as a result of refining a finite-element mesh or octree, or because a molecule breaks apart or combines with another. Such changes might also occur as a result of load-balancing, where the distribution of array elements to processors changes because of variations in processor load, for example when running on a workstation cluster. MAPS is designed to put flexible and powerful tools in the hands of application writers for these tasks as well as direct numerical tasks.

## 3. Project Description

We propose to build a new programming environment which avoids most of the restrictions imposed by the ancient array construct, allowing the production of efficient and comprehensible software which runs on the current and next generation of parallel architectures. It will be easy to learn, quick for writing and porting codes, and it will generate efficient executables.

It will be easy to learn because we use the well-understood concepts of *set* and *map* as the basic concepts to build on: these are already well understood ideas for the intended audience of computational scientists, and the conceptual

solidity of the notation used will make the constructs look familiar enough that the learning curve is as flat as possible, and it will provide natural building blocks for constructing complex modules and complete codes.

Codes will be easier to write because the concepts embedded in the language are closer to the concepts in the mathematics. By programming at the level of abstraction of a Set or a Map, the programmer is relieved of having to deal with all of the complexities and intricacies of message passing, data decomposition, race conditions, buffer allocation and so many other things provide fertile ground for bugs in current languages require but have nothing to do with the physics being simulated.

Sets and Maps in the context of an object oriented language like C++ will provide building blocks for complex codes for the same reason they provide building blocks in mathematics and in other types of programming. They are very general. They are designed from the beginning to be extensible.

They are also designed from the beginning to be efficient. Polymorphism will allow us to have different optimized implementations for a Set or a Map for different contexts. The ordering of elements in and explicit Map (an array) is not specified, but the user is guaranteed that two Maps that are aligned have the same ordering and may be, for example, added together with a vector operation. C++ operator overloading will allow us to implement lazy evaluation and run-time optimization for complex expressions and loops. On the first pass through an expression it would gather information and figure out optimizations, and on later passes it would use those optimizations.

Though the largest benefits from the MAPS system would be to entirely new codes, it is not necessary to entirely recode an existing application to benefit from this package. A major thrust of the project will be to provide a flexible means of porting existing code with the minimum of effort. A small front-end code written with MAPS can do load-balancing, parallel I/O, etc. where the array is most difficult to use, then call subroutines written with C or Fortran which encode sophisticated numerical algorithms, and have conventional data and subscript arrays as arguments.

Many of the concepts explained in the next section are available in some form in other collection-oriented parallel languages.

- A Set defined below is a generalization of the TEMPLATE construction in High-Performance Fortran, or a Shape in C*.
- A Map can be seen as a generalization and combination of arrays and functions found in most scientific languages.
- The Index-valued subscript of MAPS is a generalization of the vector-valued subscript of HPF and the prefix subscripts of C*.

These generalizations include as special cases the constructs in existing languages, and important capabilities are gained in the generalization.

## 3.1. Concepts: Sets and Maps

We propose to build a system that has as its foundation the well understood ideas of a *set* and a *map* that will allow us to escape the specifics whenever possible and write more general routines and programs. Each of these constructs will be endowed with certain general properties that will let us build powerful and efficient tools for scientific -- including parallel -- computations that will be portable to new architectures, be reusable in new simulations, and will allow codes to be written that are more understandable and easier to modify and keep up to date than current ones tend to be.

Let us define the notation *[1..N]* to be the set of integers $i$ with $1 \le i \le N$. A one-dimensional array of length $N$ may then be thought of as a map *[1..N]* $\rightarrow S$, where $S$ is a class of data objects, such as integer, real, etc. Let us think of data not as an ordered list of objects, but as a *map* from one *set* to another. Since we shall allow much more general sets than *[1..N]* and its Cartesian products, we can accommodate much more complex data structures.

For unstructured meshes, we would like to use a set whose members are chosen from a continuous space like $\mathbf{R}^3$, where **R** is the real line. Suppose we have a set of nodes of a finite element mesh, each with a datum such as the value of the field we are solving for. Rather than the conventional approach of numbering the nodes from 1 to *N*, let us explore the idea of using the set of three-dimensional coordinates directly. Given a 3D position ***r*** which is in the set, the map provides the field value associated with that position, with no subscripts involved.

Warren and Salmon's adaptive octree code[15] uses binary keys to address the cells of the octree. A cell is identified by three binary integers, one for each dimension. Each integer is a specification of how to get to the cell from the root cell, where a 0 bit means take the left branch of the tree, a 1 bit meaning the right branch. Not all possible cells are in the octree at a given time and the tree is dynamic, so it is difficult to represent this structure with conventional subscripts. We propose to construct classes that embody these ideas.

## 3.2. The Class Set

A Set is an object that contains a bunch of objects with no repeats. Mathematically, these objects can be anything, but in MAPS we restrict that somewhat: all the objects of a Set are chosen from the same class, for example all are integers, or all are real numbers, or all are employee records, or all are 4x4 matrices.

A Set can tell you the number of elements it contains, and whether a particular element is in the Set. Its representation could be explicit -- such as {2,3,5,7,11,13} -- or implicit based on some small set of data -- the range *[-20..100]*. A Set can also be a cartesian product of Sets, allowing us to build multidimensional Sets, for example:

```
Set<int> S(1,100), T(1,5);
Set<vector<int,2> > ST=Cartesian_Product(S, T);
```

Thus `S` and `T` are sets of integers, `T` is the set {1,2,3,4,5}, and `ST` is a set of integer pairs. Sets may also be made from the binary operations Union (+), Intersect (*) and Subtract (-), as long as both operands come from the same class:

```
Set<int> a(0,57), b(38,100), c(0,100);
Set<int> d = a + b;
assert(c == d);
```

These implicitly defined Sets would not be stored explicitly, but only by the specification of a range.

## 3.3. The Class Map

A Map is a method of converting a member of a Set to a member of another class, its *range* class, for example:

```
Map<float> u(ST);
```

declares that `u` is a mapping from the set `ST`, its domain, and the range class is `float`, so that the result of the mapping is a floating-point number. This is thus equivalent to the Fortran declaration

```
real u(100,5)
```

A Map takes a Set and produces another. It has a *domain* and a *range* that are each Sets. An array is a Map that is stored explicitly and like a Set it is possible to have a Map that is implicit, requiring some sort of function evaluation to look up a particular element. A Map can be an indirection for an array lookup, it can be the array itself, it can be a general transformation to the array, and it can be a linear combination or iteration involving any of the above. This is a very general purpose tool and interface with which we can construct such things as a `SparseMatrix`, an `IterativeSolver`, a `FiniteElementMesh`.

Maps may be combined with the usual operations, or may be used as arguments to functions, for example

```
Map<float> p(ST), q(ST), r(ST);
p = 1 + 3*q + sqrt(r);
```

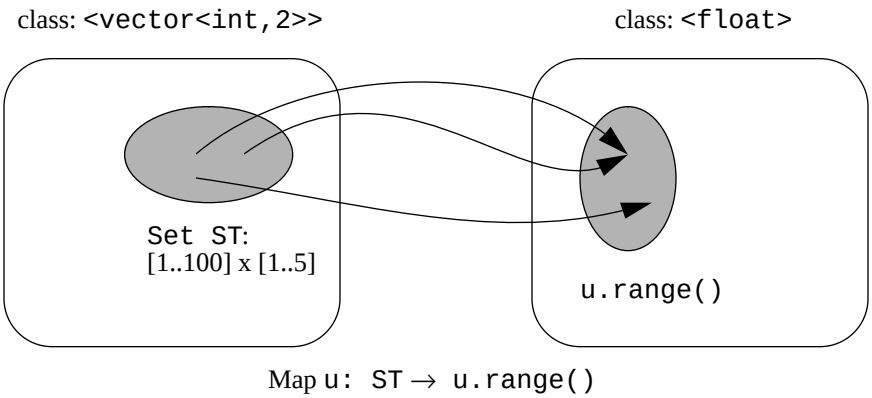


Figure 1. A map takes an element of one set and produces an element of another. A commonly used "map" is an array, but the MAPS library will allow more general domains and ranges.

which means that the value of the Map `p` for any member of `ST` is the corresponding arithmetic operation on the values of the Maps `q` and `r`.

### 3.4. How MAPS is Different

MAPS is different from existing scientific languages in several ways. Because it is built in an object oriented framework it has inherent polymorphism, meaning that, for example, a Map can be implemented in different ways to provide different functionality or efficiency in different contexts, meanwhile maintaining a consistent interface so that it may be used in standard packages. This allows, for example, a Conjugate Residual iterative solver to be handed a `HermitianLinearOperator` (a subclass of Map), and not care whether it is dense, sparse, or stored in packed or some other format, or even whether it is computed on the fly. This allows far more general and flexible numerical libraries than can currently be written. Especially on supercomputers, where the architecture can vary greatly from one to the next and the optimal data structures and algorithms must therefore vary[13], it would be a huge simplification if say, the details of how particles in a Particle in Cell code are laid out and load balanced is separated from how they are used. The result is more productive individual scientists and much more productive teams of programmers.

There is an essential new ingredient in MAPS is the dynamic creation of Sets, the implementation of which is described in [17]. A new Set can be defined as the range of a Map, which is the magic ingredient which severs the remaining ties with the brick shape of data structures in these other languages. Dynamic set creation provides the mechanism for adaptive mesh computations: when new nodes, edges and faces are added to a mesh, they are simply unioned with the existing set of mesh entities.

### 3.5. C++

C++ is sufficiently flexible and general for many programming paradigms, and has been successfully used for scientific computing[2][3][11]. MAPS would be constructed in C++, but would not require the novice user to be an initiate in the sometimes arcane lore of constructing general purpose C++ objects. We intend that MAPS will be used by a broader community as a platform to take care of parallelism and hide data structures while allowing its extension to uses we cannot see at the moment. Just as we hope that others will use our product, we shall use theirs: we intend to draw on existing libraries of C++ software for many parts of the implementation, such as a matrix library, garbage collection, load balancing, parallel sorting, and software for interfacing to existing languages.

A key goal of MAPS is wide portability, and this is a driving factor in our decision to base it on standard C++. For this project we propose no extensions to the language or changes to compilers. It is our hope and intent that this package will be usable and efficient on virtually any machine with a C++ compiler, which means virtually any computer.

## 3.6. Mask

Suppose we wish to make the Map `p` equal to the square root of all the elements of the Map `r`, but only when `r` is positive. This would be accomplished by:

```
Map<float> p = sqrt(r(r>0));
```

The object `(r>0)` is a Mask (a Map whose range class is {True, False}), and `r(r>0)` is a Map whose domain is the set such that `(r>0)` is True.

## 3.7. Index

An Index is a Map whose range is the domain of another Map; these are used to cause communication between members of Sets (and hence between processors). Thus for example in doing a finite-difference calculation with fixed boundary values, we would have two Sets: one consisting of all the nodes and another (a subset of the first) which is the interior nodes. There might be two indices, one mapping from interior points to their left hand members, the other to right hand nodes.

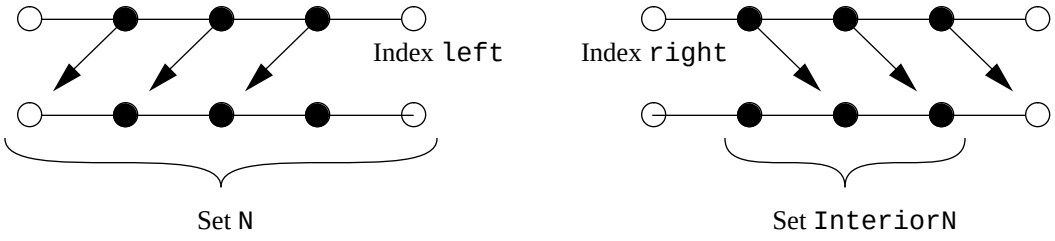


Figure 2. An Index is a Map from a Set to another. Here from interior nodes to left and right neighbors.

Suppose the nodal values are defined by a Map called `x` whose domain is the whole node set; then we could create the average of left and right values of `x` by gather operations using the Indices:

```
Map<float> xavg(InteriorN) = 0.5*(x[left] + x[right]);
```

Thus we may think of an index as a communication from one set to another, allowing data stored in the maps of one set to be used in the maps of the other; there is also a scatter-accumulate operation allowing data to be sent in the opposite direction.

## 3.8. Reduction Operations

We have seen some methods of manipulating abstract objects called Sets and Maps, but so far there is no means of extracting information from them. This is done by reduction operations, for example summing the values of a map over all elements of its domain set. Other examples of reduction operations are multiplication, logical AND and OR, maximum and minimum. It may seem churlish to restrict (at least in principle) the ability of the application program to extract data, but there are good reasons for this. This restriction not only coerces the programmer into using collection-oriented (parallel) constructs, but also gives the run-time system the freedom to store the collection in the most efficient way. An example from mathematics may illustrate this argument.

Mathematically, a vector space is a set of objects called vectors, which may be added together or multiplied by a scalar to produce new vectors. Vectors might be used represent positions in space. In order to extract physical meaning, such as an angle between two vectors, an extra component is required, an inner product. The vector itself cannot be “printed out”, but only its inner products with other vectors. In particular, we might choose some special set of “basis vectors” and represent the vector by its inner products with these, but there is nothing special about any basis set and other sets are just as good. The inner product is analogous to the reduction operation used to extract meaningful information from Maps, and the freedom to choose a basis is analogous the freedom that the MAPS run-time system has to sort Maps and distribute them among the processors of the parallel machine.

An example of a reduction operation is,

```
Set<int> S(1,100);
Map< matrix<3,3> > mat(S);
cout << reduce(sum, mat(S.data() <= 10));
```

where `mat` is a set of 100 3x3 matrices, labelled by integers from 1 to 100. The `cout` construction in C++ means to print something; in this case the sum of those matrices whose labels are not greater than 10.

The basic function of the Set is to decide if an object is a member of it, and the basic function of a Map is to take an object in the domain and return the corresponding value of the Map. Functions are provided to do these in MAPS, and are implemented in terms of reduction operations.

### 3.9. Dynamic Set Creation

In this example we illustrate the idea of dynamic set creation with an unstructured mesh example. Suppose we have a set `Elements` of elements and a set `Vertices` of their vertices, shown by the case “Linear FEM” in the Figure.

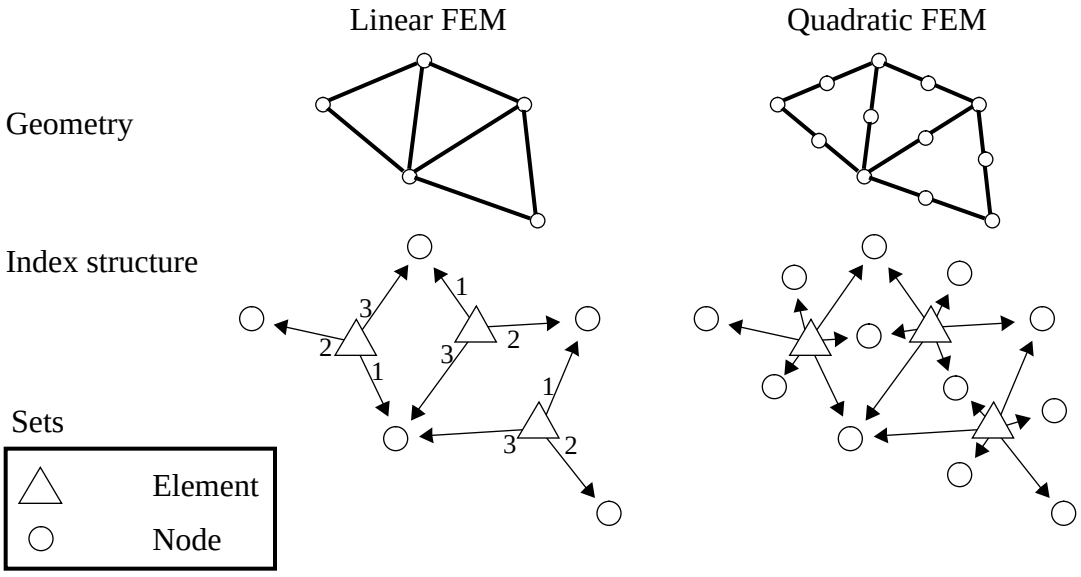


Figure 3. Extending a linear finite element mesh to a quadratic one. The new nodes at the edges are dynamically created as the range of a Map.

There are three indices connecting the elements to the vertices: `Index1` maps an element to one of the vertices, `Index2` to another, and `Index3` to the other, as shown in the Figure. We would like to run a quadratic finite element algorithm on the same mesh, meaning that many new nodes must be inserted, one at the midpoint of each edge of the mesh. This can be achieved as follows.

Three Maps are created containing the positions of the midpoints of the edges. We could name them `Mid12` for the positions of those midpoints between vertices 1 and 2 of the local numbering, `Mid23` the midpoints between nodes 2 and 3, and `Mid31` those between nodes 3 and 1. We can now express the set of midpoints as the Union of the ranges of these Maps:

```
Set<Point> Midpoints = Mid12.range() + Mid23.range() + Mid31.range();
```

where the operation + is overloaded for Sets to mean Union. At this point, we could union the new nodes with the old, interpolating any finite-element fields onto the new nodes also.

This procedure[17], done in a few lines of MAPS code, would be a serious software project in other languages. Since the code is to run on a distributed memory machine, complex message-passing is required, with many easy solutions to beckon the unwary -- which turn out not to scale well with the number of processors.

### 3.10. Connection to C and Fortran

At each stage and level of a given code, it is important to write in the appropriate language. It is vitally important that MAPS be able to interface cleanly with C and Fortran for two reasons:

- To make use of the tremendous amount of existing code in those languages.
- Low level constructs are often best expressed in a lower level language that is closer to the hardware.

MAPS would be used primarily on the higher levels of the program, expressing physics, numerics and load balancing algorithms where the clean interface and the natural separation of algorithm from implementation are important, while machine dependent modules that for example interact directly with a machine's intrinsic communication or I/O calls would be in C or Fortran for efficiency.

### 3.11. Applications

We foresee several types of applications for MAPS.

The first is in expressing types of algorithms that are complex and awkward to express in conventional array-based languages: unstructured grid generation, octree simulations, multigrid, circuit simulation, CAD tools. By allowing the programmer to express the algorithm more naturally and more independent of the messy and machine dependent details, simple algorithms can be more quickly and reliably implemented, and complex ones can be realistically constructed on schedule.

Far more general purpose mathematical libraries can be constructed. The details of grid representation and how, for example a curl is represented can be separated from the physics section of the code that specifies when a curl is to be used. A sparse matrix can be defined and passed to an iterative solver far more robustly because the matrix itself will be able to know things about itself like how to multiply it by a vector, perhaps an approximate inverse for a preconditioner, perhaps something about its spectrum. The iterative solver could then be written without depending on knowing the data formats being used.

By being able to express and compartmentalize algorithms that are awkward with today's tools it will be easier to reliably construct complex applications like chemically reacting fluid codes in complex geometries using fast implicit solvers on a variety of parallel architectures. It will allow the people on a project to more easily specialize to handling the numerics, the physics, the hardware dependencies or the overall structure of the application and thereby improve the productivity of a team of scientists.

## 4. Implementation

We describe here some overall aspects of the implementation of the MAPS library. The details of the class hierarchies and implementations are part of the proposed research, but some overall structure is clear at this time.

## 4.1. The Class Set

A Set may stored implicitly, unevaluated, or explicitly.

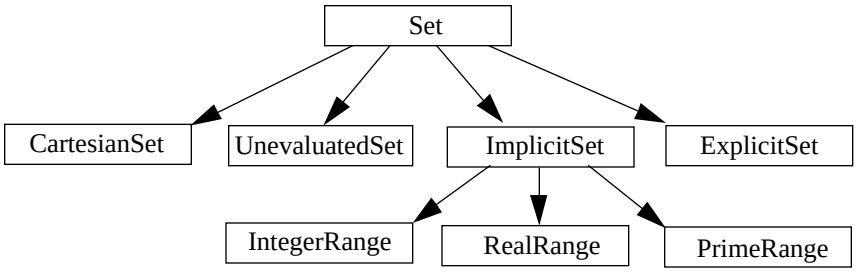


Figure 4. A schematic view of the Set class hierarchy. The various specialized subclasses of a Set share the same interface as the root class, but each provides specialized functionality.

An *implicit* Set is either a linear array Set such as *[1...10]*, which is stored as an integer lower bounds and integer upper bound, or it may be a Cartesian product of two Sets, which is simply stored as pointers to the two operands. Given this category and the Cartesian product, we can represent multidimensional arrays.

An *unevaluated* Set is one which exists briefly during expression evaluation, for example it may be a Union, Intersect or Subtract of two other Sets, stored as the relevant operator and pointers to the operands. Often a Set may be a Union of more than two Sets, such as `ABCD=A+B+C+D`; it is usually more efficient to do a Union operation just once with all four sets rather than evaluate `A+B`, then the Union of this with `C`, then the Union of this with `D`. Similarly, a Set may be a subset of another: a combination of the parent Set and a Mask.

This use of unevaluated objects is the beginnings of the idea of run-time compiling: in principle we could evaluate nothing and do no arithmetic until the application code tries to print something, at which point evaluation occurs.

An *explicit* Set is one which has been created dynamically from Maps, using an associative memory. An associative memory is a software module that takes an object from the domain class of the Set and returns an integer subscript. When a new object is stored in the memory, it is assigned the next available subscript. The associative memory is implemented as a hash table with a linked list at each table entry.

## 4.2. The Class Map

Maps may also be *implicit*, *unevaluated*, or *explicit*, and have a given Set for its domain and a type for its range. An explicit Map with a implicit domain Set like *[1...N]* is equivalent to a FORTRAN array. An implicit Map would be like a function which takes a member of a Set as its input. Note that an implicit Map with an implicit domain Set can have an infinite domain.

An unevaluated Map, like an unevaluated Set, is a temporary expression. A gather operation, for example, need not be done by copying all the data to a temporary array, but its elements may be fetched only when an evaluation is really necessary.

An explicit Map with an explicit domain Set is what we use to build things like unstructured grids. The domain is the list of elements (or nodes, or edges) and the range is a floating point number or a vector. They can be added, subtracted, multiplied etc. without reference to the domain set, enabling vectorization. The range could also be something like a pointer, allowing a Map to mean something like “the element to your left.”

An Index is an extension of this idea, and also may also be of implicit or explicit form. An implicit Index for use in a structured grid might be one mapping a grid node to the left-hand neighbor node, in which case it is stored with very little space; an explicit Index is stored as an array of subscripts into the Map arrays of the target Set, which requires an

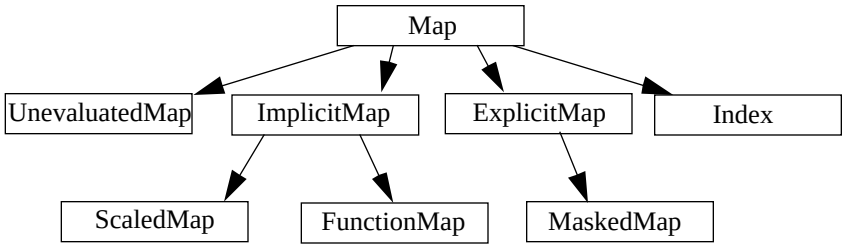


Figure 5. A schematic view of the Map class hierarchy. As with Set, the specialized subclasses of a Map share the same interface as the root class Map, but each provides specialized functionality.

indirectly addressed loop for a gather or a scatter-accumulate operation, and upon creation the Index would calculate things like communication patterns or sorts that would let it calculate the scatter-accumulate more efficiently.A Graph of Sets

The structure of a MAPS application is ia graph of Sets connected by Indices, with several Maps attached to each Set. The Sets serve as an organization for the application data, which is stored in the Map arrays. Indices link the Sets together, providing communication between different elements of the Sets. While the (explicit) Sets are stored with the less efficient methods of hash table and linked list, the bulk of computation occurs on the right side of the picture, being the computation with Map arrays and addressing with Indices. This latter part will be implemented with fast vector operations to obtain maximum efficiency.

While the implementation is in terms of arrays and subscripts for efficiency, the application code does not have access to these. This means that MAPS is free to sort arrays as it sees fit, and use local (on-chip) subscripts rather than carrying the useless baggage of global subscripts.

## 4.3. Parallelism

On a distributed memory machine, explicit Sets are distributed between the processors, possibly with cached copies on several processors, so Set elements may be classified as “real” and “copy”. When there are no Indices, computation is “embarrassingly parallel” on all the Maps attached to a given Set., and the computation occurs on the copies also. When an Index is created, it points to the local copy of the Map data: a gather operation occurs only on the “real” Set elements, followed by a communication between processor to update the copies and maintain coherency. A similar communication occurs with a scatter-accumulate operation.

But this expensive communication may not need to happen at each gather or scatter operation. We will investigate run-time dependency analysis so that when communication occurs, independent objects can be made coherent at the same time.

## 4.4. Load Balancing

We shall implement load balancing as another kind of Map, so that the domain of the Map is the Set to be distributed, and the range is the set of processors running the application. Several load balancing modules will be provided by MAPS, such as Inertial Recursive Bisection[18] for geometric objects such as meshes, and Recursive Spectral Bisection[12] for more general use. An application developer can also utilize her own knowledge of the application to load-balance in the optimal way, while gaining the benefit of being able to express it in a high level language like MAPS.

### 4.5. Input/Output

The objects that are manipulated by a MAPS program have the appearance of being a single object that happens to be distributed and the input/output formats would reflect this. It would be writable from a parallel machine and then readable on another one. A Set is intrinsically an unordered collection of objects, meaning that files containing Set and Map definitions have the same meaning if their records are arbitrarily permuted. The processors of a parallel machine can read and write records asynchronously, providing a great speedup over the traditional file of ordered records.

## 5. Bibliography


1 G. E. Blelloch and G. W. Sabot, *Compiling collection-oriented languages onto massively parallel computers*, J. Par. and Distr. Computing, **8** (1990) 119.

2 K. C. Budge, J. S. Peery and A. C. Robinson, *High-performance scientific computing using C++*, Proc. 1992 Usenix C++ Technical Conference, Protland OR 1992.

3 D. W. Forslund, C. Wingate, P. Ford, J. S. Junkins and S. C. Pope, *A distributed particle simulation code in C++*, Proc. 8th Computing in Civil Engineering Symposium, American Society of Civil Engineers, 1992, pp 514-518.

4 P. J. Hatcher and M. J. Quinn, *Data-parallel programming on MIMD computers*, MIT press, Cambridge MA, 1991.

5 K. E. Iverson, *A Programming Language*, Wiley, New York 1962.

6 K. Kennedy and C. Koelbel, *High performance Fortran Language Specification*, obtainable by anonymous ftp from titan.rice.edu in /pub/HPFF.

7 C. Koelbel and P. Mehrotra, *Compiling global name-space parallel loops for distributed execution*, IEEE Trans. Parallel and Distributed Systems, **2** (1991) 440.

8 S. R. Kohn and S. B. Baden, *An implementation of the LPAR parallel programming model for scientific computations*, p 759 in Proc. 6th SIAM Conf. on Parallel Processing, eds. R. F. Sincovic, D. E. Keyes, M. R. Leuze, L. R. Petzold and D. A. Reed, Society for Industrial and Applied Mathematics, Philadelphia PA, 1993.

9 J. R. Larus, B. Richards and G. Viswanathan, *C**: A large-grain, object-oriented data-parallel programming language*, University of Wisconsin Technical report 1126, 1992.

10 J. K. Lee and D. Gannon, *Object oriented parallel programming experiments and results*, p 273 in Proc. Supercomputing '91, ACM Press, 1991.

11 J. A. Lewis, S. M. Henry, D. G. Kafura and R. S. Schulman, *An empirical study of the object-oriented paradigm and software reuse*, SIGPLAN Notices **26** (1991) 184.

12 A. Pothen, H. D. Simon and K. P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. **11** (1990) 430.

13 S. Karmesin, *Three Dimensional Adaptive Parallel PIC Methods*, submitted to J. Comp. Physics.

14 Thinking Machines Corporation, *Getting Started in C**, Cambridge MA, 1991.

15 M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree N-body algorithm*, submitted to Proc. Supercomputing '93, ACM Press, 1993.

16 R. D. Williams, *Distributed Irregular Mesh Environment*, 1990. Documentation and source code obtainable by anonymous ftp from delilah.ccsf.caltech.edu in pub/dime.

17 R. D. Williams, *Voxel databases: a paradigm for parallelism with spatial structure*, Concurrency **4** (1992) 619.

18 R. D. Williams, *Performance of dynamic load-balancing algorithms for unstructured mesh calculations*, Concurrency **3** (1991) 457.