Machine and collection abstractions for user-implemented data-parallel programming[1]

Magne Haveraaen
Department of Informatics, University of Bergen,
P.O. Box 7800, N-5020 BERGEN, Norway

Data parallelism has appeared as a fruitful approach to the parallelisation of compute-intensive programs. Data parallelism has the advantage of mimicking the sequential (and deterministic) structure of programs as opposed to task parallelism, where the explicit interaction of processes has to be programmed.

In data parallelism data structures, typically collection classes in the form of large arrays, are distributed on the processors of the target parallel machine. Trying to extract distribution aspects from conventional code often runs into problems with a lack of uniformity in the use of the data structures and in the expression of data dependency patterns within the code. Here we propose a framework with two conceptual classes, `Machine` and `Collection`. The `Machine` class abstracts hardware communication and distribution properties. This gives a programmer high-level access to the important parts of the low-level architecture. The `Machine` class may readily be used in the implementation of a `Collection` class, giving the programmer full control of the parallel distribution of data, as well as allowing normal sequential implementation of this class. Any program using such a collection class will be parallelisable, without requiring any modification, by choosing between sequential and parallel versions at link time. Experiments with a commercial application, built using the Sophus library which uses this approach to parallelisation, show good parallel speedups, without any adaptation of the application program being needed.

1. Introduction

Programming of parallel high performance computers is considered a difficult task, and to do so efficiently may require extensive rewriting of a sequential program. One of the simpler approaches to parallel programming is the *data parallel* model, see [6] for a general introduction. Data parallel programming represents a generalisation of SIMD (Single Instruction Multiple Data) parallel programming. Early data parallel languages include Actus [35], *Lisp and CM-Fortran for Thinking Machine's Connection Machine series with up to 65 536 processors (see [22] for experience with the CM-1), MPL [28] for the MasPar series of machines, and Fortran-90 [1], which was mostly developed during the 1980s. Much of the work on data parallelism has been influenced by research on systolic arrays [30]. The popularity of data parallel programming took off with the massively parallel machines available in the late 1980s.

Task parallelism, also known as *control parallelism* or *functional parallelism*, is the notion of independent tasks communicating with each other [23]. It is intrinsically non-deterministic in nature. Task parallelism requires a much more involved semantical approach, checking that tasks rendezvous appropriately in addition to checking the sequential correctness of each task and any of the possible compositions of the tasks [33]. In contrast, data parallelism is much simpler to work with, see [21]. Data parallelism allows us to use sequential reasoning techniques when writing, reading and understanding code [16]. The programs will behave consistently on synchronous (SIMD) or asynchronous (MIMD – Multiple Instruction, Multiple Data) platforms [17].

Here we extend the use of data abstraction oriented programming to provide a framework to encompass data parallel programming. The importance of data abstraction and encapsulation has been recognised for a long time [34]. It is now supported by most programming languages, often under the terminology object-orientation and class abstraction.[2] Our idea is

[1] This investigation has been carried out with the support of the European Union, ESPRIT project 21871 SAGA (Scientific computing and algebraic abstractions) and a grant of computing resources from the Norwegian Supercomputer Committee.

[2] Based on concepts already clearly defined in the programming language Simula [12].

Scientific Programming 8 (2000) 231–246
ISSN 1058-9244 / $8.00 © 2000, IOS Press. All rights reserved

to develop a normal, sequential program using data abstractions. Then one or more of the abstractions are replaced by semantically equivalent, but data parallel versions [44]. The new configuration will create a data parallel version of the whole program without requiring any reprogramming or extensive compile-time analysis. The basic module for parallelisation is a parameterised data abstraction (also called template class or generic class), `Collection`, with user oriented methods for permutations and data updates. The `Collection` is then implemented as normal sequential code or by using a `Machine` template class. `Machine` provides concepts corresponding to the machine hardware: processors with local memory and communication channels. This may reflect a distributed memory parallel computer, but may also represent a shared memory parallel machine with hierarchical memory (local cache, local memory, off-processor shared memory, secondary store for paging, etc.). The embedding of a `Collection` onto the parallel machine is then expressed by using `Machine` in the data structure for the collection. This gives the programmer full control over the distribution and partitioning strategies. We envisage a library of `Machine` and `Collection` classes representing both sequential and parallel embeddings. Then a programmer may easily experiment with different parallelisation strategies of a program by choosing between different implementations in the library.

The programming strategy to utilise the collection classes is close to using array/collection expressions in languages like APL [24], Actus or Fortran-90. This is in contrast to a data parallel programming language like High Performance Fortran (HPF) [26,38], which is focused around loops (with explicit parallelism) over individual elements of the collections. But by making the choice of collection class implementation available to the programmer, the control over distribution of data is closer to that of HPF where this can be controlled by directives. The choice between collection class implementations does not require adaptation of the program code, and can be postponed to link time. We will refer to this as configuration time, which is when the components to make up a program are chosen, irrespectively of when this happens in the compile-link-run phases of software development. Configuration does not interact with the actual code of programs and algorithms, but is the act of choosing between (library) implementations.

Being a data parallel framework, our suggestion differs dramatically from parallel programming using message passing, e.g., the MPI library [40,41]. Message passing is associated with task parallelism programming, even though it often is used in a more restricted context when programming parallel machines. Message passing is normally added to some standard programming language, and the nature of this language will guide what kind of abstractions and programming style the programmer has to work with. However, the kind of data that can be sent across the network using message passing (typically only primitive types or contiguous array segments) will influence which abstractions are natural to work with.

This paper is organised as follows. In the next section we discuss some parallelisation strategies for the data parallel programming approach. In Section 3 we present the classes `Collection` and `Machine` of our data parallelisation framework together with a framework, Sophus, which supports the use of such abstractions. Then we provide benchmarks for parallel speedup. Finally we compare our approach with similar approaches. C++ [42] and Fortran-90 is used for code examples and benchmarks.

2. Data parallelism

The idea behind data parallel programming is that a sequential program may be parallelised by running identical copies of it synchronously as separate tasks. The data is distributed between these tasks, and a reference to non-local data causes a communication between processes. This approach can be proved to be safe in general [2]. Thus no additional reasoning, beyond that of showing it is a correct sequential program, is required to ensure the correctness of any data parallel program.

A data parallel compiler tries to extract the parallelism present in a sequential program and target it for a parallel machine. Normally the elements of array data structures, or other collection oriented data structures, are the candidates for being distributed. Distribution can be looked upon as the task of mapping the index set, the *index domain*, of the array onto the physical processors. If the mapping is a surjective function all processors will receive some work. Normally the mapping cannot be injective, as there will be more elements in the index set than there are processors on the target machine. So the index domain may have to be partitioned such that each partition is mapped to a distinct processor. According to [13], there are two philosophies to partitioning:

1. machine based: starting from the hardware, introducing virtual processors for the elements of the index domain, and

2. program based: grouping of the index domain of the data structure.

Partitioning is always subject to (1) the constraint of balancing computational load as evenly as possible between the processors and (2) minimising interprocessor data access, i.e., to minimise communication between processors, and keep communication as close as possible to the communication topology of the target machine.

Meeting these requirements requires extensive analysis of data access patterns within a program. The analysis must be performed for all data structures that are candidates for distribution, and the result must be a coherent distribution strategy for all the data of a program. It is important that data structures that interact are aligned, i.e., that their index domains are partitioned using the same strategy. Additionally, the optimal distribution of data may change throughout the program, requiring redistribution of the data as the computation proceeds. Partitioning should also take into account memory hierarchy and communication speed characteristics. Chatterjee et al. [8,9] have studied this problem and use techniques from optimisation theory to find optimal data distributions.

Though data parallelisation of a sequential program always will be correct, in the sense that the parallel versions will compute the same results as the sequential versions, there is no guarantee that the parallel versions will be efficient. While some parallel speed-up can be expected, the optimal speed-up can never be expected, as the general partitioning problem is NP-complete [27]. Thus the user normally needs to supply more information in order to achieve optimal speed-up. In HPF this is given as directives. The information may otherwise take the form of language extensions, or be interactive instructions to the compiler or run-time system, see [11] for an overview. The form of this information will appear differently depending on whether the partitioning strategy is virtual machine based or program based. Program based partitioning is the more common strategy at present. HPF and its compilers are based on this strategy [10].

A parallelising compiler for a language like HPF often treats the directives only as hints on the distribution of data. The compiler is free to distribute data in other ways it may find beneficial. This often has to do with alignment of data, where directives that conflict with the use of the data can be discovered. In such a case the compiler may rely more on the intended use of the data than on the hints it has been supplied with.

3. An abstraction based framework for data parallelism

Instead of defining a data parallel language or having the compiler look for implicit parallelism in sequential programs, we will identify abstractions, within a sequential language framework, that allow us to capture parallelism explicitly. Our data parallel framework will thus appear as sequential programming, yet provide the programmer with explicit control over the parallel distribution of the program. Reflecting the machine and program based approaches to data partitioning, we identify two collection-oriented abstractions:

1. a class `Machine` that reflects the set P of physical processors of the target machine and its communication topology, and
2. a class `Collection`, reflecting the arrays and other collection data structures declared in a program, with user-chosen index domain I and user-defined permutation patterns.

Both of these classes must be parameterised by a template class `T`. This ensures that we can place any data structure `T` on the processors of the target machine. The intention is that parallelisation of a program will be achieved by embedding the user-level abstraction `Collection` into the machine abstraction `Machine`. This will allow full programmer control over the embedding, and different embeddings for the same `Collection` will provide different parallelisation schemes for any program that can use the `Collection` abstraction.

The basic methods in the interface of the `Machine <T>` class are derived directly from hardware characteristics.

– Methods `map` to perform an argument procedure `f` on the local data of type `T` in parallel for each processor $p \in P$ of the machine, where the behaviour of `f` may depend on the processor index p, but `f` is only allowed to access data local to its processor p.
– (Several) methods that capture the structured data permutations between processors consistent with the communication network topology of the hardware.
– (Several) methods that capture unstructured data permutations which are supported by the hardware.
– Broadcast, reduction/prefix operations as supported by the hardware.

- Methods to update and read individual data values at specific processors.

The topology of a machine may take into account issues as relative access costs of hierarchical memory, not just the physical wiring between processors which traditionally is seen as the connectivity of a machine. Many of the operations above should be familiar as array operations. For example, mapping the plus operation over a pair of collections gives the elementwise array addition operator of Fortran-90. Parallelism is achieved by the `map` operations (`forall` style parallelism), which execute `f` at every processor in parallel, and the permutation operations, which communicate in parallel over the network. Access to a location at a specific processor will force sequentialisation and should be avoided.

For the Connection Machine series a `CM_Machine` abstraction will reflect the machine's hypercube architecture. The class `CM_Machine` will have the processor numbers as index domain, such as the set $\{0, 1, \dots, 65\,535\}$. The permutations will permute data between neighbouring processors for every link of the hypercube, i.e., for $i \in \{1, 2, \dots, 16\}$ the pairs of processors that differ in bit i will swap data.

For the MasPar machine series the `MP_Machine` abstraction will reflect the machine's 2-dimensional toroidal architecture. So for a 64 by 128 processor MasPar MP-2, `MP_Machine` will have the pairs $\{0, 1, \dots, 63\} \times \{0, 1, \dots, 127\}$ as index domain. The permutations will shift data between processors for every link of the collection, i.e., circularly in directions North/South or East/West, or circularly in the diagonal connections NE/SW or NW/SE. The MasPar also has a general router between arbitrary processors, but it does not perform well for massive data exchanges, and may be consciously omitted from the permutation methods of `MP_Machine`.

For modern machines like a 128 processor SGI Cray Origin 2000 or networks of (multiprocessor) workstations, like 32 SUN Ultra-10, the network topology becomes more vague. Both these configurations have specific links, but neither offers any direct means of controlling which connection is to be accessed. Thus we may think of one as a 128 node *fully connected* machine, the other as a 32 node *fully connected* machine. In both cases the index domain for `FC_Machine` will be a range of integers, and any communication pattern repeated on all processors is a legal data permutation. In these examples we have not worried about the effect of hierarchical memory.

The user level abstraction `Collection` is quite similar to `Machine`. In addition to the template class `T`, we formally include the index domain I as template, giving the abstraction `Collection<I,T>`. Each variable (or object) belonging to this class has a shape $I' \subseteq I$ and only contains data belonging to the subdomain I' of the index domain I. In C++ a rank n index[3] set I may be realised by the class `Index<`n`>` which represents all n independent integer indices $i_1, \dots, i_n$. For $n = 3$ we may define the extent I' of the indices to be $i_1 \in \{0, 1, 2, 3, 4\}$, $i_2 \in \{0, 1, 2\}$, $i_3 \in \{0, 1, 2, 3\}$ when we declare a variable `ind`.

```
int ilim[3];
ilim[0]=5; ilim[1]=3; ilim[2]=4;
Index<3> ind(ilim);
```

Constructs for defining arbitrary index sets, akin to `set` in Pascal [25], makes it possible to define non-contiguous subsets I'' of I'. We may now declare a `Collection<I,T>` variable `A` to have `double` as elements and 3 indices in the range of i_1, i_2, i_3 from above by

```
Collection<Index<3>,double> A(ind);
ilim[0]=2; ilim[1]=2; ilim[2]=2;
set(ind,ilim);
update(A,ind,5.2);
```

The last three statements set the index `ind` to $(2, 2, 2)$ and then updates the element at that position in `A` to `5.2`.

The methods for a `Collection` class are basically the same as for the `Machine` classes.

- methods `map` to perform an argument procedure `f` on the local data of type `T` in parallel for each data element $i \in I'$ of the index domain, where the behaviour of `f` may depend on the index i, but `f` is only allowed to access data local to its index i,
- (several) methods for structured and unstructured permutations of data, as needed by the user,
- methods for broadcast, reduce (with an arbitrary associative operation $\oplus$), prefix etc., as needed by the user,
- methods to update and read individual data values at specific indices $i \in I'$, and
- methods for extracting and replacing all elements corresponding to a subset I'' of I'.

[3] A rank n index set I is such that the elements of I can be identified by n independent indices $(i_1, \dots, i_n) \in I$.

The majority of the operations are only defined as needed by the user. This is because different applications have different requirements, and by tuning the interface of `Collection` to the needs of the user we lessen the demands on its implementation, making it easier to provide versions tuned for different parallel machines.

The important aspect of the `Collection` abstraction is that it promotes access to the entire structure at once, separating permutation operations from computations and element update operations. This promotes a programming style which discourages arbitrary access patterns to data elements, but any useful access pattern may of course be defined in an appropriate permutation operation. It also obliterates a construct like HPF's "independent do" directive which only has a meaning for explicitly indexed elements in loops. Instead this will be built into the requirements of relevant (unstructured) permutation operations or be handled by map operations. For example, if computations `f` and `g` are to be applied to elements indexed by disjoint subsets $J, K \subseteq I'$, respectively, then mapping an operation `p(i,x)` for index `i` and data element `x` with body `if i in J then f(x) else if i in K then g(x)` over the elements of the collection will do the trick.

The user oriented permutation operations should be restricted as much as reasonable. The more limitations that can be placed on these permutations, the better, as this will give a larger freedom of choice for the parallel embeddings.

The rest of this section will first present an approach to programming numerical software well suited for using the `Collection` abstraction. Then we will discuss implementing sequential and parallel versions of `Collection` and how to define the `Machine` class.

3.1. The Sophus library

As noted, the use of the proposed data parallel framework requires a more holistic view of programming than the elementwise manipulations commonly found. But using this is feasible. We have used it with success in the area of partial differential equations (PDEs) [19].

Historically, the mathematics of PDEs has been approached in two different ways. The applied, solution-oriented approach uses concrete representations of vectors and matrices, discretisation techniques, and numerical algorithms. The abstract, pure mathematical, approach develops the theory in terms of manifolds, vector and tensor fields, and the like, focusing more on the structure of the underlying concepts than on how to calculate with them (see [39] for an introduction).

The former approach is the basis for most of the PDE software in existence today. The latter has very promising potential for the structuring of complicated PDE software when combined with object-oriented and template class based programming languages. Some current languages that support these concepts are Ada [3], C++ [42], Eiffel [29] and GJ [7]. Languages that do not support templates, such as Fortran-90 [1] and Java [15], may also be used, but at a greater coding cost.

The Sophus library framework [18,20] provides the abstract mathematical concepts from PDE theory as programming entities. Its components are based on the notions of manifold, scalar field and tensor field, while the implementations are based on conventional numerical algorithms and discretisations. Sophus is being investigated by implementing it in C++. The framework is structured around the following concepts:

- Variations of the basic collection classes as sketched above. The `map` operations for numerical operations like `+`, `*` have been explicitly defined.
- Manifolds. These are sets with a notion of proximity and direction. They represent the physical space where the problem to be solved takes place.
- Scalar fields. These may be treated as arrays indexed by the points of the manifold with reals as data elements. Scalar fields describe the measurable quantities of the physical problem to be solved, and are the basic layer of "continuous mathematics" in the library. The different discretisation methods provide different designs for the implementation of scalar fields. A typical implementation would use an appropriate collection as underlying discrete data structure. In a finite difference implementation partial derivatives are implemented using simple shift permutations and arithmetic operations on the collection.
- Tensors. These are generalisations of vectors and matrices and have scalar fields as components. Tensors are implemented using collections with scalar fields as template arguments.
- Equation administrators. These handle the test functions for finite element methods, volumes for finite volume methods, etc., in order to build (and solve) systems of linear equations. Again, these classes are implemented using appropriate collections.

Each of the collections classes used in the abstractions above may have a different implementation.

A partial differential equation uses tensors to represent physical quantities in a system, and it provides a relationship between spatial derivatives of tensor fields and their time derivatives. Given constraints in the form of the values of the tensor fields at a specific instance in time and relevant boundary conditions, the aim of a PDE solver is to show how the physical system will evolve over time, or what state it will converge to if left by itself. Using Sophus, the solvers are formulated at the tensor layer, giving an abstract, high level program for the solution of the problem.

3.2. *Implementing the abstractions*

3.2.1. *The* `Collection` *classes*

Any given `Collection` variant may be implemented in many ways. Different sequential and distributed implementations provide different data layout, data traversal and parallelisation strategies. A configuration for a program will define which of these implementations, and hence strategies, to use. Since the collection classes are data abstractions, a new version may be implemented by the user, e.g., if a new hardware topology should require it. This also gives an opportunity for experimenting with optimisation and parallelisation strategies under full programmer control.

A sequential `Collection` is a normal data abstraction, implemented using sequential code, e.g., using the standard array structures of a programming language. The data layout in memory will be significant if issues like cache misses etc. are important for the target machine. Configuring a program from sequential collections will give a sequential program.[4] Sequential collections may be nested freely in the same way as any other data type constructor.

A parallel version of the collection classes may be achieved using a `Machine` abstraction as data structure for `Collection`. We assume that we may distribute a collection data type in isolation from the rest of the program, that it then can be nested with other data type constructors, and that the set of distributed variables in a program will give a coherent distribution of that program. These assumptions are normally satisfied.

All collection classes which satisfy the same interface and functionality are interchangeable, irrespectively of whether the implementations are sequential or parallel. Thus a program that uses `Collection` classes may be run on sequential and parallel machines without altering the program code. The key is the configuration where which `Collection` implementation to use is defined.

Consider the simple example of a matrix data structure `BlockCollection<P,J,T>`. We want to distribute it among the processors of a parallel machine as a blocked matrix implementation, where the index domain P for the blocks coincide with the index set for the processors of the target machine, and each block has index domain J. Distributing the blocks directly on the processors would correspond to the data structure `Machine<Collection<J,T>>`, where the inner class `Collection<J,T>` is sequential, i.e., is in local memory at each of the processors. This can be expressed by the C++ template class declaration

```
template
  <class P, class J, class T>
  BlockCollection
  { Machine<Collection<J,T>> B;
    ...
    public:
    void blockperm();
    void columnperm( int j );
    ...
  }
```

where `blockperm` and `columnperm` are two of possibly many permutation operations for this collection class. Assume that `blockperm` is an operation that will permute the blocks in a pattern required by the user, for instance for a block matrix multiplication. If this permutation pattern coincides with the topology of the machine, `blockperm` my be implemented using a call directly to one of `Machine`'s permutation operations `perm`.

```
template
```

[4] Ideally such a program should have the same run-time efficiency as a conventionally written program, where the data permutation patterns are intermixed with expressions. This will often not be the case, see the discussion in [14], due to current compiler optimisation technology. This technology is geared towards certain programming styles, giving large efficiency improvements to programs expressed using certain idioms, with little or no improvements on programs written using other styles. But optimisation technology is not static, and history has shown that it will adapt to new usage idioms, supporting styles which people believe are important. A discussion of recent optimisation technology improvements for C++ can be found in [36]. User controlled optimisation techniques based on abstraction oriented programming approaches are also being developed, see for example [43,14]. Thus one's fear that the intense use of abstractions may slow down a program may be confirmed by current benchmarks, but this is a situation that should change quickly when the usefulness of abstraction oriented techniques has been established.

```
  <class P, class J, class T>
  void BlockCollection<P,J,T>::
       blockperm()
  { B.perm(); }
```

The permutation operation `columnperm(int j)` which permutes column j of each block between the blocks in the pattern of the topology, can be code by first extracting the column at all processors, then permuting them, and finally replacing the permuted column on all processors.

```
template
  <class P, class J, class T>
  void BlockCollection<P,J,T>::
       columnperm(int j)
  { Machine<Collection<J,T>> C;
    map( B, extract, j, C );
    C.perm();
    map( B, replace, j, C );
  }
```

Here `extract(Collection<J,T> M, int j, Collection<J,T>& c)` is a `Collection` operation that extracts column `j` of the matrix `M` and stores it in the column variable `c`. The operation `replace(Collection<J,T>& M, int j, Collection<J,T> c)` will do the opposite, i.e., replace column `j` in `M` by `c`. These operations are performed on all processors by the `map` operations, and will, for each block at each processor, in parallel, extract and then replace the appropriate portion of the distributed variable `C`. The distributed data in `C` is permuted according to the topology of `Machine` between the mapped `extract` and `replace` operations.

The permutation patterns needed by the user will not always match exactly the communication topology of the target machine. Then the implementation of the permutation operations for the collection will require more complex expressions of communication operations. Most of these communication expressions will in any case be straight forward, since the machine topologies are adapted to the permutations likely to be needed by users. This typically includes the regular permutations needed for matrix multiplication, many linear equation solvers, fast Fourier transforms, finite difference methods, etc. At the general level, it has been shown [31] that any regular permutation pattern may be built in a small number of steps from a few basic communication operations, namely those of meshes and hypercubes. The cost of finding the optimal sequence may be fairly high, but once found, it may be coded in the implementation of a `Collection` class and freely reused.

Irregular permutation patterns may also be defined as a permutation operation on the collection class [37], irrespectively of whether such communication is directly supported by hardware. In fact, it may be beneficial to implement unstructured permutations by a sequence of structured permutations, even if there is hardware support for irregular permutations [4].

If we look closer at the problem of distributing a data type `Collection<I,T>` by the data structure `Machine<Collection<J,T>>`, for a machine with processors indexed by P, we see that we can define an injective *location mapping* $\ell : I \to P \times J$. This forces a lower bound on the size of J given I and P. Then $\ell(i) = (p, j)$ will indicate which processor $p \in P$ and which index $j \in J$ at that processor the element i of the collection `Collection<I,T>` is mapped to. This is true both when the number of data elements is larger than the number of processors (the most common case), but also when there are more processors than elements in the index domain I. In that case J may be taken to be a one-element set, which is equivalent to declaring the data structure `Machine<T>` rather than `Machine<Collection<J,T>>`. The sequential case is when the set P contains one element. Then the data structure reduces to `Collection<J,T>` and ℓ defines a reindexing of the data elements, e.g., how to map a rank n index set I into a rank 1 index set J.

The location mapping ℓ can be defined from two functions, the *distribution mapping* $d : I \to P$ and the *local memory location mapping* $e : I \to J$, by $\ell(i) = (d(i), e(i))$. The functions d and e define the distribution among processors and memory layout within a processor of the data, respectively. Such functions may be used explicitly in the implementation of the collection, or may be given implicitly by the way data of `Collection<I,T>` is addressed on `Machine<Collection<J,T>>`. It is always useful to state these functions explicitly, as this is a good documentation of the data layout. Normally one would require d to be surjective so that all processors receive data. Also, making e surjective would ensure a good utilisation of memory at each of the processors. Common strategies for d and e include address splitting, where some bits of an index $i \in I$ are chosen as processor number and some bits are chosen as location index within each processor. This gives a uniform distribution of the data elements, but may not give a uniform distribution of work load between processors. If there is some subset $I' \subseteq I$ that we expect to be heavily compute intensive, we would prefer that the distribution mapping d maps the elements of I' as uniformly as possible across the processors, and how the rest of the data is mapped is less crucial.

Equally important is taking into account the communication topology of the target machine in case of a distributed memory machine. Permutations of `Collection<I,T>` will have to be expressed as communications of `Machine`, and the more directly we may exploit the hardware topology, the more efficient our program will become.

Sequential and shared memory machines apparently have no network communication topology in the traditional sense. But these typically exhibit complex notions of memory hierarchies: various caches, non-local memory, disk storage for paging, etc. Constraints may then be put on the local memory location mapping e to utilise hierarchical memory efficiently. In practice this seems to be more difficult to control than the utilisation of processors, as the strategies used by memory management systems rarely are available to application developers. For high performance applications, cache and memory misses may be more costly than poor parallel load balancing. Currently we are forced to experiment in the dark with different memory localisation strategies, and rely on the quality of compilers to avoid unnecessary high execution costs in this area.

When building complex programs using collection classes one will often find that collection classes are nested within collection classes. This is the case for the Sophus framework, where the tensor fields are collections of scalar fields, which themselves are implemented using collections of reals. Sequential implementations of collection classes may be nested freely with sequential and parallel classes. A collection class based on a `Machine` class can be nested within another parallel collection class only if we have a nested parallel machine structure, e.g., a networked cluster of multiprocessor machines. Some approaches to nested parallelism exist at the programming language level, see [5], and may be exploited to allow such nesting, but this requires advanced handling by the compiler. Otherwise machine classes should not be nested in a program configuration, neither directly nor indirectly, hence we may only have one parallel collection class in the construction of a nested data structure. If the parallel collection class is deeply nested, we typically get fine-grained parallelism (large data structure with parallel replication of the small parts). If the parallel collection is one of the outer classes we tend to get coarse grain parallelism (a few parallel instances of large data structures).

3.2.2. The `Machine` *classes*

The `Machine` class is where the data distribution and parallelism is implemented directly in low-level, hardware oriented concepts. This is in a setting where we only have to worry about the hardware characteristics and not about data partitioning, virtualisation, user defined permutations, etc. The main implementation techniques available are

– a sequential language together with message passing akin to MPI, typical for programming MIMD machines, but many message passing libraries have also been adapted for SIMD machines, and
– a language with specific parallel constructs for the target hardware, such as the data parallel C dialect MPL for the MasPar computer series, typical for programming SIMD machines.

Although general, platform independent, parallel languages exist, these will seldom be suitable for implementing a `Machine` class, since they provide an extra layer of abstraction which destroys the close relationship with hardware that we want to achieve. The closer we are to the hardware, the better it is in the case of implementing the `Machine` classes.

Building a `Machine` class using message passing in principle opens up for all the problems of showing correctness that are associated with task parallelism. But the SPMD (Single Program, Multiple Data) restriction of the MIMD model provides assumptions which greatly simplify correctness arguments in our framework. An SPMD execution of a program makes identical copies of the same program code and runs them concurrently, one on each processor. The only place where communication takes place in our framework is within the permutation, broadcast/reduce, update and read operations of `Machine`. If certain simple guidelines on writing program code is obeyed, these will be executed synchronously in the SPMD model. So we just have to ensure that the communications are correct within each of these communication operations. Thus the reasoning for correctness is simplified from showing correctness for all executions of a collection of tasks [33], to showing correctness for all executions of each of the permutation procedures. This can be shown once for each implementation of a machine abstraction – and need not be repeated for each program being developed as for general task parallelism. The reason is that operations on `Machine<T>` are collective operations performed by all SPMD processes on the same arguments, see [2] for details. In contrast, message passing calls in the MIMD style, even if executed as SPMD processes, are generated unrestricted on each processor, requiring extensive proofs to guarantee correct matching of communication at all stages of the program.

The SPMD implementation of `Machine` for MIMD execution on P processors may be obtained by the following.

– Use `T` as data structure for `Machine<T>`.
– `map` executes the argument procedure on the data structure.
– Permutations are implemented by appropriate message passing send/receives or permutations.
– Broadcast and reduce/prefix operations are implemented as supported by the message passing system. If not supported, they will not be provided by the machine abstraction.
– Individual requests for updates and reads are directed to the appropriate processor using broadcast or send/receive operations.

With this strategy all data is declared on all processors, which gives a P-indexed set of data, and all operations are automatically run in parallel on all processors. The reason is that the SPMD execution style implicitly distributes all variables by replicating the whole program. This allows the compiler to be ignorant about the parallel structure of the program. Only the `Machine` implementation is aware of this fact, so only the abstract variables defined via the machine class will be truly distributed, the others will be replicated. This also goes for the computations, which will be replicated for all sequential variables, and only the distributed variables will benefit from the parallel execution. Since the most compute-intensive variables normally are declared using `Machine`, the repeated computation, in parallel, of the replicated variables should represent a small fraction of the total time. This then gives a good speed-up, see the benchmarks in Fig. 8 for the application SeisMod.

The communication operations for `Machine` must be provided for any type `T`, not only the predefined types. This may add some technical difficulties in the implementation of `Machine`, as any composite, non-primitive type, may have to be broken down into primitive components, transmitted, and then be rebuilt by the receiver.

The technique of overlapping communication operations with computation to reduce waiting time is difficult to achieve in the proposed framework. It would require splitting the `Machine` communication operations into: communication initiate and communication close suboperations. The splitting would have to be propagated up to the collection abstractions, which would violate the uniformity of the interface for different collection class implementations, especially between sequential and parallel versions. The splitting would also make program development much more difficult. A way to handle this is through a tool like CodeBoost [14] which could utilise the algebraic properties of the communication suboperations to restructure the program when such overlap is required.

Programming `Machine` classes does not run into the problem of communication between tasks if an SIMD language is used, where all operations are synchronous. SIMD language attributes will typically identify data as distributed (indexed by the processors) or sequential (replicated on the processors or stored on a frontend processor). Permutations, broadcasts, reduce/prefix operations will be supported in the SIMD language if available on the machine. Likewise with access to individual elements. So an SIMD language will have direct constructs supporting all operations of a `Machine` class. These operations should also be at a sufficiently low level to achieve efficiency at the hardware level. A drawback may be that many machines, hence their SIMD languages, have restrictions on which data types may be moved between processors in atomic operations, or which associative operations are supported by reduction/scan operations. The general forms must then be implemented for the machine classes. More seriously, some SIMD languages have restrictions on how to declare distributed data. The problems this entails and how to handle this is outside the scope of this paper, see [17] for some indications.

3.2.3. The Sophus collection classes

The abstraction oriented approach to parallelisation has been used in Sophus. There the index domain I for the `Collection` is an n-dimensional discrete Cartesian index space, so the `Collection` can be considered a rank n array, i.e., an array with n indices.

A `Collection` class with shift operations, for an arbitrary distance along any of the n dimensions, as the permutations, has been defined. It has been implemented in both a sequential version and a parallel version for the fully connected machine. The MPI message passing library provides the operations needed for `FC_Machine`, as well as more user level operations, and was used directly for the collection code. Since MPI is fairly portable, this gave easy access to parallel implementations on both SGI Cray Origin 2000 and a network of 32 SUN Ultra-10 connected by ethernet.

Ideally we should have used language and communication operations as low-level and close to the hardware as possible. For pragmatic reasons we did not do this in our pilot implementation. The implementations of the collection data structure and methods have been designed with the specific requirements from the Sophus application programs SeisMod in mind [19].

The sequential implementation of `Collection` uses a rank 1 array with index domain J as main data structure. Then the `map` operations just traverse these arrays linearly, providing a very simple but fairly efficient implementation, since we get uninterrupted loops over the data. The mapped numerical operators were explicitly defined. The shift operations rearrange data in the array, and the choice of local memory location mapping $e : I \to J$ is essential for the speed of these operations. The layout is such that shifts in the lower dimensions move large consecutive blocks of data, while the blocks become smaller and more fragmented as the dimension number increases. Shifts are performed with equal frequency in all dimensions, but since the number of dimensions normally is low this generally gives good behaviour.

The parallel implementation of `Collection` also uses a rank 1 array as main data structure for the efficient implementation of `map`. Since a fully connected architecture does not put any constraints on data movement, the choice of distribution mapping $d : I \to P$ is independent of such topological constraints. The data sets to be stored in `Collection` were typically rectangular, so splitting the data sets into equally large blocks by cutting the rectangle in pieces along the longer side gave a simple implementation with a balanced distribution of data and reasonably low communication costs. The local memory location mapping $e : I \to J$ uses the same principle as in the sequential case, taking care that contiguous blocks of data are used for interprocessor permutations. Other usage patterns would benefit from other distributions of data. In general, communication costs will be lowest if the data blocks distributed are as square as possible.

4. Benchmarks

Test runs of the pilot implementation of this framework approach to parallelisation were made on a network of 32 SUN Ultra-10 (UltraSPARC-IIi processor with 300 MHz clock and 99.9 MHz bus and 128 MByte internal memory) connected by a 10MHz ethernet network and on an SGI Cray Origin 2000 with 128 processors. On both computing resources the speed of both the sequential and parallel implementations were measured, and a comparison with similar computations in sequential Fortran-90 has been included. To show certain effects of hierarchical memory, the tests were run using rank 3 array data sets with $8{*}8{*}8 = 512, 16{*}16{*}16 = 4096, 32{*}32{*}32 = 32\,768$, $64*64*64 = 262\,144$ and $128*128*128 = 2\,097\,152$ data elements each. These cube shapes do not reflect the actual usage patterns of the applications the pilot implementation was tuned for. The tests are for single precision floating point numbers (4 bytes), thus yielding collection variables with data sizes of 2 kB, 16 kB, 131 kB, 1.0 MB and 8.4 MB each, respectively. Five measurements were taken for each tabular entry, and the median value (at processor zero) has been used. The machines are multi-user machines, and the tests were run during normal operation, but at a time with reasonably low load. On the SUN network the test runs would typically be the only compute-intensive jobs running, while the Cray had idle processors and a load between 50 and 80 during most of the tests.

The test programs were compiled using commands equivalent to those in Fig. 1, where g, which ranges through 8, 16, 32, 64 and 128, designates the number of data elements in each direction, and p, which ranges through 1, 2, 4, 8, 16, 32, 64 (as relevant), is the number of processors the code is intended for. The flag `_Par_` controls when the configuration uses the parallel implementation of `Collection`. No code in the program, or other classes used, is dependent on this flag.[5]

The first set of tests is on the speed of the shift operation.

- Direction 1: data movement internal for each processor, large contiguous segment moved at each processor.
- Direction 2: data movement internal for each processor, many small segments moved at each processor.
- Direction 3: in the parallel case, contiguous segments communicated between processors. In the sequential case, the data segments moved are more fragmented than those for direction 2.

[5] This is not fully true in our test programs, as we have included (in the main program) some code that prints out the configuration, number of processors being used and the execution speed of the tests. This code is sensitive to the configuration flags. The rest of the code is written without regards to these falgs.

```
sun4-seq> CC -I. -fast AbstractMatrix-tes-c++.C -DGRIDSIZE=g
sun4-par> CC -I. -fast AbstractMatrix-tes-c++.C -DGRIDSIZE=g -D_Par_ -DNOPROCESSORS=p
sun4-f90> f90 -fast -fixed AbstractMatrix-tes-f90.F -DGRIDSIZE=g -DITERATIONS=1
cray-seq> CC -I. -Ofast AbstractMatrix-tes-c++.C -DGRIDSIZE=g
cray-par> CC -I. -Ofast AbstractMatrix-tes-c++.C -DGRIDSIZE=g -D_Par_ -DNOPROCESSORS=p
cray-f90> f90 -cpp -Ofast AbstractMatrix-tes-f90.F -DGRIDSIZE=g -DITERATIONS=1
```

Fig. 1. Compilation commands for the benchmarks.

SUN4 processors	Mflshps, direction 1		Mflshps, direction 2		Mflshps, direction 3	
8^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	74	74	55	55	21	21
2	41	20	88	44	0.49	0.25
4	71	18	136	34	0.37	0.09
seq	76		56		21	
16^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	98	98	70	70	28	28
2	61	30	178	89	2.4	1.2
4	143	36	350	88	1.6	0.41
8	271	34	622	78	0.87	0.11
seq	63		60		29	
32^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	51	51	52	52	36	36
2	66	33	96	48	5.3	2.6
4	92	23	206	52	6.3	1.6
8	338	42	624	78	3.4	0.43
16	695	44	1187	74	2.9	0.18
seq	53		53		38	
64^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	32	32	32	32	25	25
2	67	33	91	45	11	5.4
4	159	39	209	52	13	3.2
8	281	35	384	48	6.9	0.86
16	646	40	816	51	6.3	0.39
32	1208	38	1587	50	5.0	0.16
seq	32		31		25	
128^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	29	29	32	32	29	29
2	53	27	67	33	21	10
4	102	26	131	33	23	5.7
8	203	55	263	33	16	2.0
16	509	32	745	47	15	0.92
32	964	30	1584	50	8.4	0.3
seq	29		32		29	

Fig. 2. Test runs showing speed of shift operation on a network of SUN Ultra-10 workstations

The results are given in Figs 2 and 3. The speed is given in million of floating point numbers shifted per second, Mflshps.

If shifting of data in all directions is an important aspect of an algorithm, we find that small problems should definitely be run in single processor mode. Larger problems may have an optimal number of processor to run on, see the effects on direction 3 (“total” column). For 64^3 elements the optimum seems to be between 8 and 16 processors on the Cray. Adding more processors probably will decrease the total throughput of the program due to the relative dominance of the

Cray processors	Mflshps, direction 1		Mflshps, direction 2		Mflshps, direction 3	
8^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	59	59	53	53	34	34
2	60	30	88	44	8.2	4.1
4	100	25	140	35	7.6	1.9
seq	60		55		35	
16^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	76	76	73	73	52	52
2	104	52	142	71	24	12
4	200	50	270	68	23	5.8
8	367	46	495	62	20	2.5
seq	76		73		52	
32^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	28	28	67	67	47	47
2	93	47	135	68	23	12
4	235	59	322	81	31	7.7
8	502	63	657	82	34	4.3
16	899	56	1264	79	37	2.3
seq	28		66		44	
64^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	27	27	68	68	55	55
2	110	55	146	75	29	14
4	220	55	294	73	44	11
8	441	55	585	73	55	6.9
16	799	50	1170	73	66	4.1
32	1313	41	2769	87	47	1.5
seq	27		67		54	
128^3 elts.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	17	17	28	28	18	18
2	36	18	82	41	24	12
4	146	37	289	72	43	11
8	360	45	597	75	59	7.3
16	483	30	1194	75	79	4.9
32	508	16	2529	79	74	2.3
64	374	5.8	5002	78	7.8	0.12
seq	20		36		27	

Fig. 3. Test runs showing speed of shift operation on a SGI Cray Origin 2000 multiprocessor

slow interprocessor communication as the number of processors increase. The same effect is visible for the SUNs. Due to the much higher communication costs over an ethernet network, the ideal number of processors is lower than for the Cray. With 64^3 elements, around 4 processors seems best.

For the mapped arithmetic operations the story seems different. These have no communication and scale nicely when parallelised. See Figs 4 and 5 for the addition of two matrices and Figs 6 and 7 for a scalar multiplied with a matrix. Computation speed is measured in million of floating point operations per second, MFLOPS. For the Fortran-90 tests we used 3-dimensional arrays and the built-in array operations. What is startling is the relationship between C++ and Fortran-90 execution speeds. The C++ implementation is ahead on small arrays. On the SUN, Fortran-90 speeds only approach C++ speeds for larger data sets, while on the Cray, Fortran-90 manages to surpass C++ speeds for the larger data sets. This may imply that the Fortran compiler on the Cray does a better job of aligning data with memory blocks.

Also noticeable on both platforms are irregular changes in execution speed per processor as the data size increases. Matrix-scalar multiplication, which involves only one large collection, seems to outperform addition, which involves two large collections. These changes signals two things. The first is the problem of hierarchical memory, and the interplay between data size and cache/memory misses. This will give lower

SUN4	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
Add	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	49	49	27	27	27	27	17	17	17	17
2	97	49	54	27	53	26	34	17	34	17
4	190	48	193	48	108	28	81	20	68	17
8	—		393	49	215	27	211	26	134	17
16	—		—		658	41	423	26	269	17
32	—		—		—		857	27	740	23
seq	49		27		26		17		17	
f90	6.8		10		11		8.7		8.5	

Fig. 4. Test runs showing speed of 3-dimensional matrix addition on a network of SUN Ultra-10 workstations

Cray	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
Add	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	36	36	36	36	28	28	25	25	11	11
2	71	35	72	36	55	27	51	26	10	10
4	138	34	144	36	110	27	101	25	60	15
8	—		288	36	288	36	207	26	129	16
16	—		—		579	36	435	27	430	27
32	—		—		—		882	28	880	28
64	—		—		—		—		1096	17
seq	40		40		26		22		13	
f90	19		35		47		26		31	

Fig. 5. Test runs showing speed of 3-dimensional matrix addition on a SGI Cray Origin 2000 multiprocessor

speeds as data size increases. The other effect is that of longer arrays allowing more efficient use of vectorisation and loop optimisation technology. This should give higher execution speeds as data size increases. This effect also takes place on each processor in the parallel case, where often the per processor speed increases as the number of processors increase. Thus it seems very important to be able to control data alignment. Currently no explicit language or library construct for this is available. Only indirect techniques, such as reversing the ordering of declarations or changing the initialisation ordering for the data arrays, can be used to achieve this kind of alignment of data.

Random load distributions on multi-user machines may have a marked influence on the speed of programs using large data sets. This was observed for the tests. Normally the speeds of a series of test runs would vary by no more than 10–15%, but on the Cray the individual measurements for the test case of 128^3 elements would vary far more than this. In extreme cases a factor of more than 2 was observed between the slowest and fastest test runs.

To confirm the efficiency on a real program, the parallelisation strategy was tested on the application SeisMod, a collection of programs for seismic simulations[6] written using the Sophus library. The tests were run on the isotropic and transverse isotropic versions of the program. The isotropic case means that the rock the seismic wave traverses has identical physical properties in all directions. In the transverse isotropic case the rock will have different properties in the vertical and the horizontal directions. The latter case has a more complex mathematical formulation than the former, requiring more computations per timestep. The program was run for 1000 time-steps, and distributed on varying numbers of processors. Speed-up is measured as sequential simulation time divided by parallel simulation time. Two parallelisation strategies were chosen. The first, shown in Fig. 8, parallelises the scalar field, the inner collection class of Sophus, which gives a fine-grained parallelisation. Here we see that the execution time of the main simulation decreases with the number of processors, but with a noticeable efficiency drop-off starting at 8 processors. This is probably due to the loss of efficiency in shifting data between processors when

[6] SeisMod is marketed by UniGEO a.s., Thormøhlensgt. 55, N-5008 Bergen, Norway.

SUN4	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
Mult	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	49	49	49	49	34	34	27	27	27	27
2	98	49	100	50	68	34	66	33	54	27
4	192	48	198	50	139	35	137	34	108	27
8	—		378	47	386	48	277	35	217	27
16	—		—		779	49	553	35	513	32
32	—		—		—		111	35	1084	34
seq	49		49		35		27		27	
f90	15		30		32		27		27	

Fig. 6. Test runs showing speed of 3-dimensional matrix–scalar multiplication on a network of SUN Ultra-10 workstations

Cray	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
Mult	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.	Total	Per proc.
1	43	43	43	43	38	38	38	38	19	19
2	85	42	86	43	77	39	77	38	55	27
4	167	42	172	43	181	45	154	39	141	35
8	—		343	43	345	43	308	39	299	37
16	—		—		690	43	619	39	299	37
32	—		—		—		1435	45	1230	38
64	—		—		—		—		2451	38
seq	51		52		39		39		26	
f90	28		71		71		61		50	

Fig. 7. Test runs showing speed of 3-dimensional matrix–scalar multiplication on a SGI Cray Origin 2000 multiprocessor

SeisMod, fine	Isotropic			Transverse isotropic		
Cray proc.	Init.	Simulate	Speed-up	Init.	Simulate	Speed-up
1	7s	1899s	0.9	11s	2687s	1.1
2	17s	1031s	1.6	27s	1538s	1.9
4	28s	543s	3.0	44s	894s	3.3
8	40s	331s	5.0	56s	504s	5.8
16	56s	257s	6.4	68s	347s	8.5
seq	5s	1641s	1	9s	2948s	1

Fig. 8. Execution times for application SeisMod with parallelisation of scalar fields (fine grain).

SeisMod, coarse	Isotropic			Transverse isotropic		
Cray proc.	Init.	Simulate	Speed-up	Init.	Simulate	Speed-up
1	6s	1957s	0.8	10s	3067s	1.0
2	14s	2527s	0.7	19s	3737s	0.8
seq	5s	1648s	1	9s	3061s	1

Fig. 9. Execution times for application SeisMod with parallelisation of tensor fields (coarse grain).

the number of processors increase. The initialisation times increase dramatically with the number of processors, but will amount to a minor part of total execution times for normal simulations. The second case, Fig. 9, shows parallelisation of the tensors fields, the outer collection class in Sophus, giving a much coarser grain. The tensor fields have between 2 and 6 components, and provides a less balanced distribution of the computational effort. We see this in an increase in execution time when the number of processors increase.

5. Conclusions

Data parallel programming coupled with data abstraction provides a powerful framework for the development of application programs that easily port between sequential and parallel machines. Porting does not require any reprogramming, just a selection of the appropriate parallelisation module from a library. It also allows the programmer to develop new parallel modules, either for moving onto a new architecture, or for application specific parallel optimisations. This requires that the parallel module may be nested at any level of the application program. This is automatically satisfied for a normal compiler and SPMD execution as supported by, e.g., MPI [40].

To fully utilise this framework for data parallel programming it must be coupled with programming styles such as that of Sophus [18–20] for coordinate free numerics. This is a style that provides a natural focus on permutations and operations mapped for all data, rather than on individual data accesses and loops. A collection abstraction suited for parallelisation may be used at many levels in an application, providing ample opportunities for different parallelisation strategies.

Experiments show that this approach scales nicely with the number of processors and provides a good speed-up compared to the sequential case if the classes selected for parallelisation contain many data items to be distributed. In Sophus this would typically be the scalar field classes, which implement the discretisation techniques for the numerical solvers. Similar speed-up results are shown by Ohta [32]. He proposes to use the scalar field class as the unit of parallelisation directly, rather than using a parallelisable collection to implement it. Programming a parallel version of a scalar field class directly gives more technical details to take into account at one time than our proposal does. It also disregards the possibility for other sources of data parallelism within an application.

A data parallel collection class proved to be fairly easy to implement, and yet obtain a speed that is not too far from equivalent Fortran-90 speed. For a comparison of efficiency differences between C++ and Fortran see [36]. We still expect a certain loss of efficiency due to the extra layer of code introduced by classes `Collection` and `Machine`. But at the same time these classes have clearer algebraic properties than evident in traditional code. These properties may be utilised by high-level optimisation tools such as CodeBoost [14], perhaps regaining more speed than that being lost.

Machines such as the SGI Cray Origin 2000 and software packages such as MPI camouflage the underlying connectivity of the hardware. Though a useful simplification for many purposes, the framework we propose here would allow programmers to take advantage of the low-level communication topology without sacrificing program structure. Thus a means of directly accessing the hardware topology for the purpose of implementing `Machine` classes would be useful for further research on our approach. A similar problem appears in the use of hierarchical memory structure of high performance machines, whether this be swapping between memory and disk or multilevel caches on a processor. This is presently beyond programmer control, but has a great influence on program efficiency. We would like to investigate closer how to control and utilise such memory structure from a high level perspective.

Acknowledgements

We would like to thank the following people for their contributions to this work: Ivar Velle, Steinar Søreide, who did most of the parallel implementations and helped with an earlier draft of this paper, Hans Munthe-Kaas and André Friis.

References

[1] J.C. Adams, W.S. Brainerd and J.T. Martin, *Fortran 90 Handbook: Complete ANSI/ISO Reference*, Intertext Publications, 1992.

[2] C. Bareau, B. Caillaud, C. Jard and R. Thoraval, Correctness of automated distribution of sequential programs, in: *PARLE'93 – Parallel architectures and languages Europe*, volume 694 of *Lecture Notes in Computer Science*, A. Bode, M. Reeve and G. Wolf, eds, Springer, 1993, pp. 517–528.

[3] D. Barstow, ed., *The Programming Language Ada – Reference Manual*, Springer, LNCS 155, 1983.

[4] P.E. Bjørstad and R. Schreiber, Unstructured grids on SIMD torus machines, in: *The 1994 Scalable High Performance Computing Conference*, E.L. Jacks, ed., IEEE, 1994.

[5] G.E. Blelloch, S. Chatterjee, J.C. Harwick, J. Sipelstein and M. Zagha, Implementation of a portable nested data-parallel language, *Journal of Parallel and Distributed Computing* **21**(1) (1994), 4–14.

[6] L. Bougé, The data parallel model: A semantic perspective, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 4–26.

[7] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, Making the future safe for the past: Adding genericity to the Java programming language, in: *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications* (*OOPSLA '98*), 1998.

[8] S. Chatterjee, J.R. Gilbert and R. Schreiber, Optimal evaluation of array expressions on massively parallel machines, *ACM transactions on programming languages and systems* **17**(1) (1995), 123–156.

[9] S. Chatterjee, J.R. Gilbert, R. Schreiber and S.-H. Teng, Automatic array alignment in data-parallel programs, in: *ACM Symposium on Principles of Programming Languages POPL'93*, 1993, pp. 16–28.

[10] F. Coelho, C. Germain and J.-L. Pazat, State of the art in compiling HPF, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 104–133.

[11] P. Crooks and R.H. Perrott, Language constructs for data partitioning and distribution, *Scientific Programming* **5**(1) (1995), 59–85.

[12] O.-J. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA 67 Common Base Language*, (Vol. S-2), Norwegian Computing Center, Oslo, 1968.

[13] J.-L. Dekeyser and P. Marquet, Supporting irregular and dynamic computations in data parallel languages, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 197–219.

[14] T. Dinesh, M. Haveraaen and J. Heering, An algebraic programming style for numerical software and its optimisation, *Scientific Programming* **8**(4) (2000), 247–259.

[15] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[16] Y.L. Guyadec and B. Virot, Sequential-like proofs of data-parallel programs, *Parallel Processing Letters* **6**(3) (1996), 415–426.

[17] M. Haveraaen, Comparing some approaches to programming distributed memory machines, in: *The 6th Distributed Memory Computing Conference Proceedings*, volume II Software, Q.F. Stout and M. Wolfe, eds, IEEE Computer Society Press, 1991, pp. 222–227.

[18] M. Haveraaen, Case study on algebraic software methodologies for scientific computing, *Scientific Programming* **8**(4) (2000), 261–273.

[19] M. Haveraaen, H.A. Friis and T.A. Johansen, Formal software engineering for computational modeling, *Nordic Journal of Computing* **6**(3) (1999), 241–270.

[20] M. Haveraaen, V. Madsen and H. Munthe-Kaas, Algebraic programming technology for partial differential equations, in: *Norsk Informatikk Konferanse – NIK'92*, A. Maus and F. Eliassen et al., eds, Tapir, Norway, 1992, pp. 55–68.

[21] M. Haveraaen and N.A. Magnussen, Why parallel programming is easy, and CSP programming is hard, in: *Norsk Informatikk Konferanse – NIK'90*, B. Kirkerud et al., eds, Tapir, Trondheim, Norway, 1990, pp. 183–192.

[22] W.D. Hillis and G.L. Steele Jr., Data parallel algorithms, *Communications of the ACM* **29**(12) (1986), 1170–1183.

[23] C. Hoare, Comunicating sequential processes, *Communications of the ACM* **21**(8) (1978), 666–677.

[24] K.E. Iverson, *A Programming Language*, John Wiley and Sons, Inc., 1962.

[25] K. Jensen and N. Wirth, *Pascal – User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*, Springer, 1974.

[26] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr. and M.E. Zosel, eds, *The High Performance Fortran Handbook*, Scientific and Engineering Computation Series, MIT Press, Cambridge Massachusetts, 1994.

[27] M.E. Mace, *Memory storage patterns in parallel processing*, volume 30 of *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, Boston, MA, 1987.

[28] MasPar Computer Corporation, *MasPar Parallel Applications Language* (*MPL*), (version 1.00), MasPar Computer Corporation, Sunnyvale, California, 1990.

[29] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1992.

[30] W. Moore, A. McCabe and R. Urquhart, eds, *Systolic Arrays*, Adam Hilger, Bristol and Boston, 1987.

[31] H. Munthe-Kaas, Routing k-tile permutations on parallel computers, in: *Norsk Informatikk Konferanse – NIK'94*, M. Haveraaen and B. Jæger et al., eds, Tapir, Norway, 1994, pp. 133–150.

[32] T. Ohta, Design of a data class for parallel scientific computing, in: *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, Y. Ishikawa, R.R. Oldehoeft, J.V. Reynders and M. Tholburn, eds, Springer, 1997, pp. 211–217.

[33] S. Owicki and D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Communications of the ACM* **19**(5) (1976), 279–285.

[34] D.C. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM* **15**(12) (1972), 1053–1058.

[35] R.H. Perrott, A language for array and vector processors, *ACM transactions on programming languages and systems*, **1**(2) (1979), 177–195.

[36] A.D. Robison, C++ gets faster for scientific computing, *Computers in Physics* **10**(5) (1996), 458–462.

[37] J.H. Saltz, G. Agrawal, C. Chang, R. Das, G. Edjlali, P. Havlak, Y.-S. Hwang, B. Moon, R. Ponnusamy, S.D. Sharma, A. Sussman and M. Uysal, Programming irregular applications: Runtime support, compilation and tools, *Advances in Computers* **45** (1997), 105–153.

[38] R.S. Schreiber, An introduction to HPF, in: *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, G.-R. Perrin and A. Darte, eds, Springer, 1996, pp. 27–44.

[39] B. Schutz, *Geometrical Methods of Mathematical Physics*, Cambridge University Press, 1980.

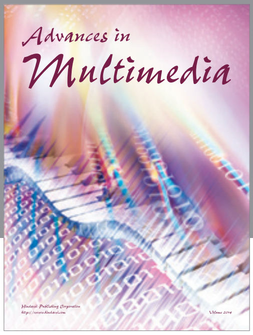
[40] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker and J. Dongarra, *MPI – The complete reference*, MIT Press, Cambridge, Mass., USA, 1996.

[41] J.M. Squyres, B. Saphir and A. Lumsdaine, The design and evolution of the MPI-2 C++ interface, in: *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, Y. Ishikawa, R.R. Oldehoeft, J.V. Reynders and M. Tholburn, eds, Springer, 1997, pp. 57–64.

[42] B. Stroustrup, *The C++ Programming Language*, (3rd ed.), Addison-Wesley, 1997.

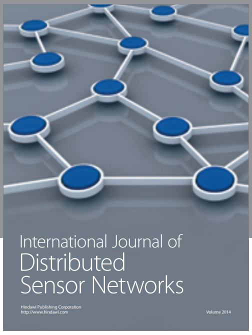
[43] T.L. Veldhuizen and M.E. Jernigan, Will C++ be faster than Fortran, in: *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, Y. Ishikawa, R.R. Oldehoeft, J.V. Reynders and M. Tholburn, eds, Springer, 1997, pp. 49–56.

[44] I. Velle, Abstrakte datatyper på parallelle maskiner, Master's thesis, Universitetet i Bergen, P.O.Box 7800, N-5020 Bergen, Norway, August 1993.

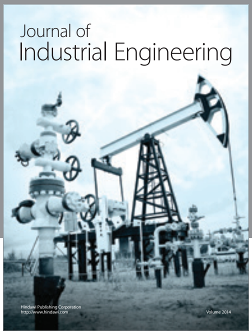


Advances in Multimedia

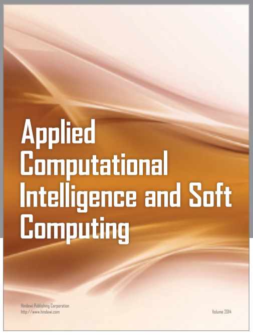
The Scientific World Journal



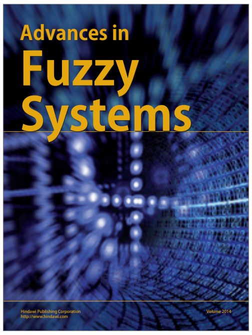
International Journal of Distributed Sensor Networks



Journal of Industrial Engineering



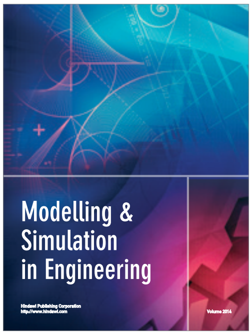
Applied Computational Intelligence and Soft Computing



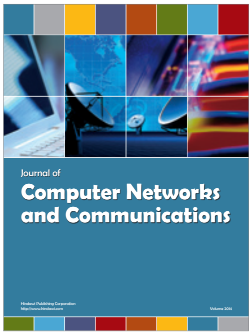
Advances in Fuzzy Systems

Hindawi

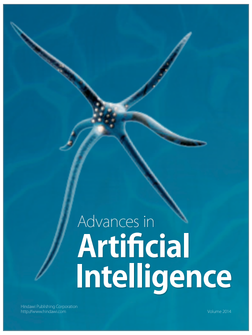
Submit your manuscripts at
http://www.hindawi.com



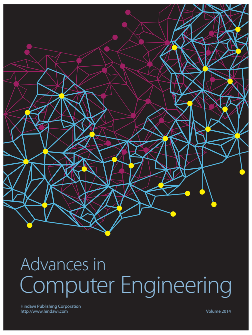
Modelling & Simulation in Engineering



Journal of Computer Networks and Communications

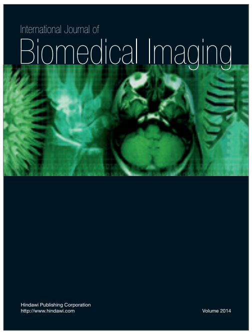


Advances in Artificial Intelligence

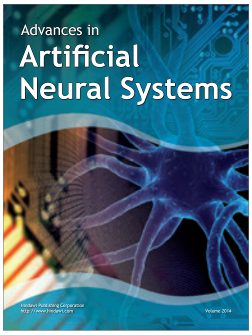


Advances in Computer Engineering

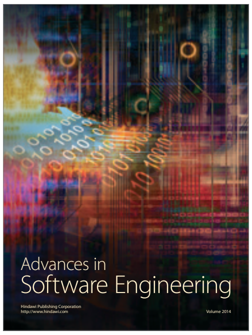
International Journal of Computer Games Technology



International Journal of Biomedical Imaging

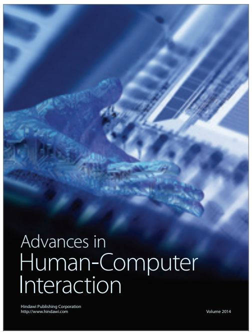


Advances in Artificial Neural Systems

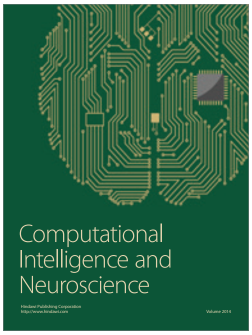


Advances in Software Engineering

Journal of Robotics



Advances in Human-Computer Interaction



Computational Intelligence and Neuroscience

International Journal of Reconfigurable Computing

Journal of Electrical and Computer Engineering