Reports in Informatics no. 162, December 1998, pp. 23, ISSN 0333-3590.
Department of Informatics, University of Bergen, N-5020 BERGEN, Norway

Abstractions for Programming Parallel Machines*

Magne Haveraaen†

Abstract

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 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 defining a data abstraction of hardware communication and distribution properties, which may be readily used in the implementation of collection classes. This gives a programmer high-level access to the important parts of the low-level architecture. Experiments with a commercial application, built using the Sophus library which uses this approach to parallelisation, show good parallel speed-ups, without any modification 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, also known as the SPMD (Single Program, Multiple Data) model, see [Bou96] for a general introduction. Data parallel programming evolved from research on systolic arrays [MMU87], and represents a generalisation of SIMD (Single Instruction Multiple Data) parallel programming as supported, e.g., by the language MPL [Cor90] for the MasPar series of machines. The popularity of data parallel programming took off with the massively parallel machines available in the late 1980's, see [HJ86] for experience with Thinking Machine's Connection Machine CM-1 with up to 65 536 processors.

Task parallelism, also known as *control parallelism* or *functional parallelism*, is the notion of independent tasks communicating with each other ([Hoa78]). 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 ([OG76]). In contrast, data parallelism is much simpler to work with, see [HM90]. Data parallelism allows us to

*This invesitgation 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.

†University of Bergen, Department of Informatics, Høyteknologisenteret, N-5020 BERGEN, Norway.

use sequential reasoning techniques when writing, reading and understanding code ([GV96]). The programs will behave consistently on synchronous (SIMD) or asynchronous (MIMD – Multiple Instruction, Multiple Data) platforms ([Hav91]).

Here we extend the use of data abstraction oriented programming to encompass data parallel programming. The importance of data abstraction and encapsulation has been recognised for a long time ([Par72]). It is now supported by most programming languages, often under the terminology object-orientation and class abstraction[1]. Our idea is 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 ([Vel93]). 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 (or template class in C++ terminology [Str97]), `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. 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 such embeddings, allowing a programmer to easily experiment with different parallelisation strategies of a program by varying its configuration between different implementations in the library.

This paper is organised as follows. In the next section we discuss some parallelisation strategies for the data parallel programming apporach. In Section 3 we present classes `Collection` and `Machine` together with a framework, Sophus, which supports the use of such abstractions, discuss the validity of our parallelisation approach, and provide benchmarks for parallel speed-up. Finally we compare our approach with similar approaches in the conclusion. C++ 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, [BCJT93]. Thus no additional reasoning, beyond that of showing it is a correct sequential program, is required to ensure the correctness of any specific parallel program developed according to this strategy.

A data parallel compiler tries to extract the parallism present in a sequential program, and target it for a parallel machine. Normally 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 using a surjective function. Typically 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. According to [DM96], there are two philosophies to partitioning:

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

[1] Based on concepts already clearly defined in the programming language Simula ([DMN68]).

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

Partitioning is always subject to the constraint of balancing computational load as evenly as possible between the processors and minimising interprocessor data access, i.e., to minimise communication between processors, keeping communication as close as possible to the communication topology of the target machine. This requires extensive analysis of data access patterns within the 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 be subject to constraints such as memory hierarchy and communication characteristics.

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. The situation is rather such that although some parallel speed-up can be expected, the optimal speed-up can never be expected, as the general partitioning problem is NP-complete (see [Mac87]). Thus the user normally needs to supply more information in order to achieve optimal speed-up. In High Performance Fortran (HPF), [Sch96], 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 [CP95] for an overview. The form of this information will appear differently depending on whether the partitioning strategy is virtual processor based or index domain based. Index domain partitioning is the more common strategy at present. HPF and its compilers are based on this strategy ([CGP96]).

The compiler 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 Abstractions for distributing data

One technique for data parallelism is to let a compiler analyse a sequential program where the intended use of a variable has to be discovered. Instead we will supply the user with abstractions that allows this to be expressed explicitly. Reflecting the machine and program based approaches to data partitioning, we will provide 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,
- methods `perm` to permute data between processors consistently with the communication network topology of the machine, and
- methods to update and read individual data values at specific processors.

These operations have been known as collection operations already in APL ([Ive62]), and they should be familiar as array operations. For example, mapping the plus operation over a pair of collections gives the elementwise matrix + operator of Fortran-90 ([ABM92]). Parallelism is achieved by the `map` operations, which execute `f` at every processor in parallel, and the `perm` operations, which communicate in parallel over the network. Access to a location at a specific processor will force sequentialisation and should be avoided. Of course other collection oriented methods, such as reduction operations or the parallel prefix/the scan operation, can be included in the list. The above operations are however sufficient to demonstrate our approach.

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 $\{0, 1, \ldots, 65\,535\}$. The permutations will permute data between neighbouring processors for every link of the hypercube, i.e., for $i \in \{1, 2, \ldots, 16\}$ the pairs of processors that differ in bit i will swap data.

For the MasPar machine series the `MP_Machine` abstraction will reflect its 2-dimensional toroidal architecture. So for a 64 by 128 processor MasPar MP-2, `MP_Machine` will have the pairs $\{0, 1, \ldots, 63\} \times \{0, 1, \ldots, 127\}$ 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 the diagonal connections NE/SW or NW/SE may be used. The MasPar also has a general router between arbitrary processors, but this 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 (multi-processor) 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 an 8 node *fully connected* machine 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.

The user level abstraction `Collection` is quite similar to `Machine`. In addition to the template class `T`, we also 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.

- 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,
- methods `perm` to permute data between indices 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'.

The important aspect of the `Collection` abstraction is that it promotes access to the entire structure at once, separating permutations from element updates. It also discourages arbitrary access patterns to data elements, but any useful access pattern may of course be defined in an appropriate `perm` operation. The ability to extract data for subdomains $I' \subseteq I$ is important in that it provides a systematic way to operate on a subset of I. Likewise the permutation operations should be restricted as much as possible. For most practical problems, such as matrix multiplication, linear equation solvers, fast fourier transforms, finite difference methods, etc., it has been noted in [MK94] that permutations corresponding to those of meshes and hypercubes is all that is needed in the formulation of the algorithms. The more limitations that can be placed on the user-level 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 `Collection` using the target `Machine`, before we discuss the validity of the approach. Finally we supply benchmarks on some operations and compare this to sequential versions written in C++ and Fortran-90.

3.1 The Sophus library

As noted, the use of a `Collection` class requires a more holistic view of programming than the elementwise manipulations commonly found. But using `Collection` is a feasible approach, and we have investigated its use in the area of partial differential equations (PDEs). PDEs are being used to describe many different and important problems. This ranges from seismic simulations for oil exploration purposes via flow of liquids for injection moulding and coating, to properties of electronic circuites.

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, while the abstract, oure 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 [Sch80] 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 [Bar83], C++ [Str97], Eiffel [Mey92] and GJ [BOSW98]. Languages that do not support templates, such as Simula [iSSG87], Fortran-90 [ABM92] and Java [GJS96], may also be used, but at a greater coding cost.

The Sophus library ([HMMK92] is an early presentation) 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 the conventional numerical algorithms and discretisations. Sophus is implemented in C++. It is structured around the following concepts:

- Basic collection structures as sketched above. The `map` operations for numerical operations like `+`, `*` have been explicitly implemented.
- 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 shifts 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.

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 the application programs – 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

Sequential and distributed versions of `Collection` allow for adapting parallelisation strategies and the tuning of data structure layout in memory by choosing appropriate configurations for a program. Since the collection classes are data abstractions, the versions may be implented by the user. This gives an opportunity for experimenting with new optimisation and parallelisation strategies under full user control.

A sequential `Collection` is a normal data abstraction, implemented using sequential code, for instance using the standard array structures of a programming language. Configuring a program from sequential collections will of course give a sequential program.[2] Sequential collections may be nested freely in the same way as any other data type constructor.

[2] 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 [DHH98], due to the present optimisation technology. Compilation optimisation 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++

A parallel version of the collection classes may be achieved using the `Machine` abstraction as data structure for `Collection`. The discussion here asumes that we may distribute a collection data type in isolation, but that it then can be nested with other data type constructors and that the set of distributed variables available to a program will give a coherent distribution of that program[3].

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 each block has index domain J, and the index domain P for the blocks coinside with the index set for the processors of the target machine. 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();
  ...
}
```

where `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 coinsides with the topology of the machine, `blockperm` my be implemented using a call directly to `Machine`'s `perm` operation.

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

If we instead need a permutation operation `columnperm(int j)` that permutes column j of each block between the blocks in the pattern of the topology, we can code this 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 Collection<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

can be found in [Rob96]. User controlled optimisation techniques based on abstraction oriented programming approaches are also being developed, see for example [VJ97, DHH98]. 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.

[3] This issue is discussed in Section 3.3.

`map` operations, and will, for each block at each processor, in parallel, extract and then replace the appropriate portion of the distributed variable of columns, `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 `perm` 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. At the general level, it has been shown that any regular permutation pattern may be built in a small number of steps from a few basic communication operations ([MK94]). 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.

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[4] 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 them explicitly, as this is a good documentation of the parallelisation. 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, is mapped is less crucial. Equally important is taking into account the communication topology of the target 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. Likewise constraints may be put on the local memory location mapping e to utilise hierarchical memory efficiently. This seems to be more

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

difficult in practice than utilising processors, as the strategies used by memory management systems rarely are known 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 unecessary 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 library, 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, but `Machine` classes should only be nested 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 [BCH$^+$94], and may be exploited. Otherwise machine classes should not be nested in a program configuration, neither directly nor indirectly.

3.2.2 The machine classes

The `Machine` class is where the data distribution and parallelism is implemented using 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 either to use a sequential language together with a message passing library such as MPI ([SOHL$^+$96, SSL97]), or to use an existing language with specific parallel constructs for the target hardware, such as the data parallel C dialect MPL ([Cor90]) for the MasPar computer series. The former is typical for programming MIMD machines, but many message passing libraries have also been adapted for SIMD machines. The latter is typical for SIMD machines. Although general, platform independent, parallel langauges exist, these are seldom suitable for implementing a `Machine` class, since they provide an extra layer of abstraction which destroys the close relationship with hardware that we need to achieve.

Building a `Machine` class using a message passing library in principle opens up for all the problems of task parallelism. But the SPMD framework provides assumptions and restrictions which greatly simplify correctness arguments. The only place where communication takes place is within the `perm`, update and read operations of `Machine`, and since these will be executed synchronously, we just have to ensure that the communications are correct within each communication operation. Thus the reasoning for correctness, such as that presented in [OG76], is limited to all executions of each of these procedures, and has to be proved once for each implementation of a machine abstraction – not for all executions of every program being developed as for task parallelism. The reason is that operations on `Machine<T>` are collective operations performed by all SPMD processes with the same arguments, see [BCJT93] for details. In contrast, message passing calls in the MIMD style, even if executed as SPMD processes, are generated per processor, requiring extensive proofs for correct matching of communication at all stages of the program.

An SPMD program execution on a MIMD machine normally takes place by running several identical copies of the program, each copy on its own processor. The SPMD implementation of `Machine` for MIMD execution may be obtained by the following implementation of `Machine`.

- use `T` as data structure for `Machine<T>`.
- `map` executes the argument procedure on the data structure.

- `perm` uses permutations or appropriate send/receives from a message passing library such as MPI.
- individual requests for updates and reads is directed to the appropriate processor using broadcast operations.

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 truely 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 represent a small fraction of the total time, this normally gives a good speed-up, see the benchmarks in Figure 8.

Programming SIMD machine classes does not run into the general problem of communication between tasks if an SIMD language is used. In the data parallel C dialect MPL ([Cor90]) for the MasPar computer series, e.g., the parallel distribution of a variable is signalled by giving it the attribute `poly`. Any built-in operation applied to a poly variable is run in parallel on all processors, thus MPL provides mapped versions of all C language operations. Communication of data is done using specific `perm` operations, or operations accessing variables at specified processors. But in our programming approach we must be able to nest `Machine` abstraction arbitrarily deep within a data structure. MPL requires that `poly` only be applied at the outermost level of variable declarations, thus violating our basic assumption for the use of abstractions. Neither was it possible to turn a user-defined procedure from a sequential to a poly procedure without explicitly giving `poly` attributes to all its parameters and local variables. These drawbacks can be circumvented by having a pre-compiler restructure the code[5], but this shows that the underlying design assumptions of the parallel aspect of MPL is incompatible with modern object-oriented and abstraction based programming practices in general. This is a situation that may be typical for SIMD languages.

3.2.3 The Sophus collection classes

The abstraction oriented approach to parallelisation has been used in the Sophus library. There the index domain I for the `Collection` is a n-dimensional discrete Cartesian index space, so the `Collection` can be considered a rank n array, i.e., an array with n indices. Typically n is 2 or 3, but higher values may occur. The permutations of the `Collection` are shift operations for an arbitrary distance along any of the n dimensions. The `Collection` class is implemented in both a sequential version and a parallel version for the fully connected machine. The MPI message passing library ([SOHL$^+$96, SSL97]) provides the operations needed for `FC_Machine`, as well as more user level operations, and was used directly. 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 an ethernet. The implementations of the collection data structure and methods have been designed with the usage requirements from the Sophus application programs in mind.

[5]The needed transformation rules are described in Section 3.3.

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, since they are heavily used in the rest of Sophus. 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 topological constraints. The data sets used for `Collection` where typically rectangular, so splitting the data sets into equally large blocks by cutting the rectangle in pieces along the longer side gives 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.

3.3 Validity of approach

The validity of an abstraction based approach to data parallel programming requires both semantic notions for arbitrary data type constructions, and sequential and parallel execution models for the final programs. The data type constructors relate to each other and obey certain algebraic laws that we will explore. The relationships are given by relations $\cong$ and $\Rightarrow$. The relation $\cong$ means that if there is a program that performs some computation on the data structure on the left, then we may transform that into a program which computes the same results using the data structures on the right, and visa versa. $\cong$ is an equivalence relation on data structures. The weaker relationship $\Rightarrow$ means a program for the left side may be transformed to a program for the right, but not necessarily the other way round. If we have $S \Rightarrow T$ and $T \Rightarrow S$ then $S \cong T$. Changing the data structure according to the rules will typically break abstraction boundaries, because direct access to components that otherwise should be encapsulated are needed. This is not problematic for a compiler or a code transformation system, but is not consistent with abstraction oriented programming styles and good software engineering practices.

The standard building blocks for data structures on a computer are primitive types, records and arrays. The primitive types H are integers, characters, floats, etc. as recognised by hardware instructions or presented by a programming language. The record constructors $\times$ takes two types S and T and returns a type $S \times T$. Records may be declared explicitly, such as the C++ `struct` declaration, or they arise implicitly by the declaration of a parameter list, or local variables, for a procedure. The data type constructors array A takes a type T as argument and returns a type $A\langle T\rangle$. We simplify the presentation by playing down the role of the index domain by not mentioning it as a parameter to A. A non-distributed data type is said to be *sequential*. A data type constructor applied to a sequential type gives a sequential type. All the primitive types H are sequential. A (sequential) `Collection` class should be thought of as an array, and it will obey the rules for A.

There is a conversion from the array constructor to the record constructor,

$$A\langle T\rangle \quad\cong\quad T\times\ldots\times T, \tag{1}$$

given that the product contains the same number of elements as the size of the index domain of the array. This is a two-way transformation, since we may construct an array with the same number of elements as components in the record. Record construction is associative and commutative and distributes over arrays, and arrays are idempotent, thus

$$\begin{aligned}
(T\times T')\times T'' &\cong T\times(T'\times T''), &(2)\\
S\times T &\cong T\times S, &(3)\\
A\langle S\times T\rangle &\cong A\langle S\rangle\times A\langle T\rangle, &(4)\\
A\langle A\langle T\rangle\rangle &\cong A\langle T\rangle, &(5)
\end{aligned}$$

where of course idempotency implies a correspondence on the index domains. We also have rules

$$\begin{aligned}
T &\Rightarrow T\times T, &(6)\\
T &\Rightarrow A\langle T\rangle, &(7)
\end{aligned}$$

which say that we may replicate a data item, but that we cannot expect to merge data variables.

Data abstractions gives the programmer the power to create arbitrary data type constructors C, taking types $T,\ldots,T'$ and returning a type $C\langle T,\ldots,T'\rangle$, e.g., $C\langle T,T',\ldots,T''\rangle = A\langle T\rangle\times H\times T'\times\ldots T''$. Such data type constructors will have to be built using the basic data type constructors and other programmer defined data type constructors.

Theorem 3.1 *For any user-defined data type constructor C the following laws hold*

$$\begin{aligned}
C\langle T,\ldots,S\times S',\ldots,T'\rangle &\Rightarrow C\langle T,\ldots,S,\ldots,T'\rangle\times C\langle T,\ldots,S',\ldots,T'\rangle, &(8)\\
C\langle A\langle T\rangle,\ldots,A\langle T'\rangle\rangle &\Rightarrow A\langle C\langle T,\ldots,T'\rangle\rangle. &(9)
\end{aligned}$$

Proof If C is defined using only A and $\times$ give it nesting level 0. If C is constructed using other user-defined data type constructors, give C a nesting level one more than the maximum nesting level of its component data type constructors. Since a data type constructor cannot be circularaly defined, C must have a finite nesting level. Any data type constructor with nesting level 0 satisfies (8) due to (2), (4) and (6). If C is constructed using other data type constructors where (8) holds, then (8) also holds for C. Hence (8) holds for any nesting level. Given (8) then (9) holds by (1). □

To see that we cannot have an equivalence (8) or (9) consider the case where $C\langle T\rangle = H$. The theorem allows us to push the type constructer C inwards in a data type expression, ending up with either A or $\times$ as the outermost data type constructor.

We nay now introduce parallel distributing versions of the record, $\times_p$, and the array, A_p, data type constructors. The number of components they apply to must reflect the number of processors on the parallel hardware, and they describe how each component of the data structure is distributed onto the local memory of each processor. If nested parallelism is not allowed, these data type constructors must not be nested. By analogy with the sequential case we get the following transformation rules.

$$\begin{aligned} A_p\langle T\rangle &\cong T \times_p \ldots \times_p T, & (10)\\ T \times_p \ldots \times_p S \times_p S' \times_p \ldots \times_p T' &\cong T \times_p \ldots \times_p S' \times_p S \times_p \ldots \times_p T', & (11)\\ T &\Rightarrow T \times_p \ldots \times_p T, & (12)\\ T &\Rightarrow A_p\langle T\rangle. & (13)\end{aligned}$$

Associativity of records, sequential rule (2), is not relevant since the number of components for the parallel record is fixed by the hardware. Rule (11) states that the order of components for parallel distribution may be interchanged (commutativity). Rules (12) and (13) state that we may replicate a computation for a sequential variable on all processors. The idempotency rule for arrays is changed to a rule which allows us to introduce a parallelism corresponding to the data structure equivalence between `Collection<`$J \times P$`,T>` and `Machine<Collection<`J`,T>>`,

$$A\langle T\rangle \cong A_p\langle A\langle T\rangle\rangle, \tag{14}$$

where we introduce parallel distribution at the outermost level.

The abstraction oriented approach to data parallelism is contingent on the rule,

$$A_p\langle S\rangle \times A_p\langle T\rangle \cong A_p\langle S \times T\rangle, \tag{15}$$

which says that two independently distributed parallel variables is equivalent to the parallel distribution of a record structure.

Also we need the rules (8) and (9) to hold in the parallel version,

$$C\langle T, \ldots, A_p\langle S\rangle, \ldots, T'\rangle \Rightarrow A_p\langle C\langle T, \ldots, S, \ldots, T'\rangle\rangle, \tag{16}$$

otherwise we cannot build parallel distributed data structures from data structures with parallel components. This rule can be shown to hold using the same technique as for (9), provided parallel data structures are allowed as components of any data type constructor.

Rule (16) subsumes rules like $A\langle A_pT\rangle \Rightarrow A_p\langle AT\rangle$. Idempotency of A_p corresponds to flattening of parallelism. In cases where nested parallelism and virtual processors are allowed this may be meaningful, but normally a rule like (14) is more appropriate.

The rules (10)–(16) have a direction which allows us to move the parallel data constructors from inside to the outside of a type expression. Moving A_p outwards may force replication of variables, such as the parallelisation of sequential types in $S \times A_p\langle T\rangle \Rightarrow A_p\langle S\rangle \times A_p\langle T\rangle \cong A_p\langle S \times T\rangle$.

We now turn to the problem of what parallel data structure distribution is needed for parallel execution on the hardware architectures we are considering. A MIMD model of a SPMD program has a set of prosessors, each which execute copies of the same program in parallel on a distributed data structure. Thus MIMD execution requires us to transform the data structure of a program to the form $A_p\langle T\rangle$, where each program is only aware of the sequential execution on T. This obviuosly follows from rules (1)–(16), which hold for MIMD machines.

The SIMD machine model is a bit more complex. As in the case of MasPar, we will assume a controlling processor with sequential data structure S running in parallel with a massively parallel unit with distributed data structure $A_p\langle T\rangle$. This gives a SIMD data structure of the

```
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
```

Figure 1: Compilation commands for the benchmarks

form $S \times_p A_p\langle T\rangle$. Again this obviuosly follows from rules (1)–(16). These rules hold for SIMD machines and MasPar, but (16) does not hold in the MasPar language MPL since it is illegal to nest parallel distribution inside a data type constructor.

Thus we have shown that programs written using `Collection` and `Machine` abstractions may be translated for both SIMD and MIMD execution.

3.4 Benchmarks

Test runs of the abstraction oriented approach to parallelisation were made on a network of 32 SUN Ultra-10 (UltraSPARC-IIi processor with 300MHz clock and 99.9MHz bus and 128MByte 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 where 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. The tests are for single precision floating point numbers (4 bytes), thus yielding collection variables with data sizes of 2kB, 16kB, 131kB, 1.0MB and 8.4MB each, respectively. Five measurments 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 where 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 where compiled using commands equivalent to those in Figure 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. This tunes the storage space required by the test program to the actual data sizes used.

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.

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	

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

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	

Figure 3: Test runs showing speed of shift operation on a SGI Cray Origin 2000 multiprocessor

SUN4 Add	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
	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	

Figure 4: Test runs showing speed in MFLOP of 3-dimensional matrix addition on a network of SUN Ultra-10 workstations

Cray Add	8^3 elts.		16^3 elts.		32^3 elts.		64^3 elts.		128^3 elts.	
	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	

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

The results are given in Figures 2 and 3. The speed is given in million of floating point numbers shifted per second, Mflshps. If shifting of data in each direction 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. For 64^3 elements this seems to be around 8 to 16 processors on the Cray, where adding more processors seems to decrease the total throughput of the program.

For the arithmetic and map functions the story seems different. These have no communication and scale nicely when parallelised. See Figures 4 and 5 for the addition of two matrices and Figures 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 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 noticable are the changes in execution speed as the data size increases. These changes

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	

Figure 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	

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

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 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 construct for this is available, and only indirect techniques such as the reversing of the ordering of the declarations or splitting initialisation in pieces across the data arrays, can be used. Also random load distributions on multi-user machines may have a marked influence on the speed of programs using large data sets. This was observed on the test data. 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 effect 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

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

SeisMod, fine Cray proc.	Isotropic			Transverse isotropic		
	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

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

SeisMod, coarse Cray proc.	Isotropic			Transverse isotropic		
	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

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

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 horisontal 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 where chosen. The first, shown in Figure 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 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, Figure 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.

4 Conclusions

Data parallel programming coupled with data abstraction provides a powerful combination 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 MPI ([SOHL+96]). For SIMD compilation a language like MPL ([Cor90]) has language restrictions prohibiting this, though there is no fundamental reason for such a restriction.

To fully utilise this approach it must be coupled with programming styles such as that of Sophus ([HMMK92]) for coordinate free numerics. This is a style that provides a natural focus on permutations and operations mapped across data, rather than on individual data accesses and loops. The collection abstraction suited for parallelisation may be used at many levels in an application, providing ample oppurtunities 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. This requires that the classes selected for parallelisation contain many data items to be distributed. In Sophus this would typically be the scalarfield classes, which implement the discretisation techniques for the numerical solvers. Similar speed-up results are shown in [Oht97]. Ohta proposes to use the scalar field class as the unit of parallelisation directly, rather than using a parallelisable collection to implement it. When programming a parallel version this gives more technical details to take into account than our proposal does. It also disregards the possibility for other sources of data parallelism within an application.

The basic parallel modules in our approach 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 [Rob96].

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 abstraction methodology we propose here would allow programmers to take advantage of the communication topology without sacrificing program structure. Thus a means of accessing the hardware topology for this purpose 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.

Aknowledgements

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

[ABM92] J. C. Adams, W. S. Brainerd, and J. T. Martin. *Fortran 90 Handbook: Complete ANSI/ISO Reference.* Intertext Publications, 1992.

[Bar83] D. Barstow, editor. *The Programming Language Ada - Reference Manual.* Springer LNCS 155, 1983.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Harwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[BCJT93] Cyrille Bareau, Benoît Caillaud, Claude Jard, and René Thoraval. Correctness of automated distribution of sequential programs. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE'93 – Parallel architectures and languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 1993.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip 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.

[Bou96] Luc Bougé. The data parallel model: A semantic perspective. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 4–26. Springer, 1996.

[CGP96] Fabien Coelho, Cécile Germain, and Jean-Louis Pazat. State of the art in compiling HPF. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 104–133. Springer, 1996.

[Cor90] MasPar Computer Corporation. *MasPar Parallel Applications Language (MPL), version 1.00.* MasPar Computer Corporation, Sunnyvale, California, 1990.

[CP95] P. Crooks and Ron H. Perrott. Language constructs for data partitioning and distribution. *Scientific Programming*, 5(1):59–85, 1995.

[DHH98] T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimisation. Technical Report SEN-R9844, CWI, Amsterdam; NL, December 1998.

[DM96] Jean-Luc Dekeyser and Philippe Marquet. Supporting irregular and dynamic computations in data parallel languages. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 197–219. Springer, 1996.

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language*, volume S-2. Norwegian Computing Center, Oslo, 1968.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GV96] Yann Le Guyadec and Bernard Virot. Sequential-like proofs of data-parallel programs. *Parallel Processing Letters*, 6(3):415–426, 1996.

[Hav91] Magne Haveraaen. Comparing some approaches to programming distributed memory machines. In Quentin F. Stout and Michael Wolfe, editors, *The 6th Distributed Memory Computing Conference Proceedings*, volume II Software, pages 222–227, 10662 Los Vaqueros Circle; PO Box 3014; Los Alamitos, CA 90720–1264; USA, 1991. IEEE Computer Society Press.

[HJ86] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HM90] Magne Haveraaen and Nils A. Magnussen. Why parallel programming is easy, and csp programming is hard. In Bjørn Kirkerud, Ellen Brox, Reidar Conradi, Marit Jenssen, Khalid Azim Mughal, Jon Natvig, and Kai A. Olsen, editors, *Norsk Informatikk Konferanse – NIK'90*, pages 183–192. Tapir, Trondheim, Norway, 1990.

[HMMK92] Magne Haveraaen, Victor Madsen, and Hans Munthe-Kaas. Algebraic programming technology for partial differential equations. In Arne Maus, Frank Eliassen, et al., editors, *Norsk Informatikk Konferanse – NIK'92*, pages 55–68. Tapir, Norway, 1992.

[Hoa78] C.A.R. Hoare. Comunicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[iSSG87] Standardiseringskommissionen i Sverige (SIS) and SIMULA Standards Group, editors. *Data processing - programming - SIMULA*. Standardiseringskommissionen, Sweden, 1987. Svensk standard SS 63 61 14.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, inc., 1962.

[Mac87] Mary 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.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MK94] Hans Munthe-Kaas. Routing k–tile permutations on parallel computers. In Magne Haveraaen, Bjørn Jæger, Siri Bjorvand, Randi Karlsen, Stein Krogdahl, Arne Maus, Sigurd Meldal, and Syvert Nørsett, editors, *Norsk Informatikk Konferanse – NIK'94*, pages 133–150. Tapir, Norway, 1994.

[MMU87] Will Moore, Andrew McCabe, and Roddy Urquhart, editors. *Systolic Arrays*. Adam Hilger, Bristol and Boston, 1987.

[OG76] S. Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[Oht97] Takashi Ohta. Design of a data class for parallel scientific computing. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 211–217. Springer, 1997.

[Par72] D. C. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Rob96] Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10(5):458–462, 1996.

[Sch80] Bernard Schutz. *Geometrical Methods of Mathematical Physics.* Cambridge University Press, 1980.

[Sch96] Robert S. Schreiber. An introduction to HPF. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 1996.

[SOHL+96] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI – The complete reference.* MIT Press, Cambridge, Mass., USA, 1996.

[SSL97] Jeffrey M. Squyres, Bill Saphir, and Andrew Lumsdaine. The design and evolution of the MPI-2 C++ interface. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 57–64. Springer, 1997.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3d edition, 1997.

[Vel93] Ivar Velle. Abstrakte datatyper på parallelle maskiner. Master's thesis, Universitetet i Bergen, Høyteknologisenteret; N–5020 BERGEN; Norway, August 1993.

[VJ97] Todd L. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 49–56. Springer, 1997.