Processing Sets on a SIMD Machine

Alberto Baudino¹, Giancarlo Colla¹, Giuseppe A. Marina¹ & Giancarlo Succi²
¹DIST-Università di Genova - e-mail: {elia.joy}@dist.unige.it
²LII - Studi Aziendali Università di Trento e-mail: giancarlo.succi@lii.unitn.it

Abstract

Writing programs that really exploit the advantages of parallel computers is not easy. The programmer must know their architecture details and choose an algorithm that is suited to the kind of parallelism implemented by the machine. Imperative languages greatly feel the effects of the above mentioned problems. They lay completely upon the programmer the responsibility of choosing the proper algorithm and of implementing it. Declarative languages do not define the operation sequence needed to solve a problem, but they define the problem itself. It is up to compiler and executor to divide job into different processes and to distribute data on machine processors. Therefore they seem to be well suited to be implemented on parallel architectures. For this reason we have chosen to implement a subset equational language (SEL) on the Connection Machine 2, because the set data structure has a high degree of implicit data parallelism. This paper describes in details the mechanism by which sets are processed by the Connection Machine 2.

1 Introduction

Nowadays we witness the development of data parallel and process parallel architectures. They have a great computational power, but it is difficult writing programs that really exploit it. The reasons that can explain this situation are the following. First of all, only programmers that know architecture details of parallel computer are able to achieve the best performance from them. Second, the choice of adequate parallel algorithms is also very important to exploit the parallelism of the machine. For example a keypoint fundamental to develop a real good and optimized parallel software is the assignment of data and processes to the machine processors. A good programmer is able to minimize the execution overhead that inter-process communication and shared variables access may cause. At the opposite side, a beginner programmer can not only develop an inefficient code, but also to slow down the performance of all the programs running on that host. A good solution is to identify a set of programming languages that both help programmers to develop parallel software, and minimize damages that unoptimized software solutions can cause. Unfortunately the programming languages that are most used are the imperative ones, they greatly feel the effect of the above-mentioned problems. In fact these languages declare explicitly the operations that the processors have to do to reach the solution (i.e. it is defined the control of the program).

This approach lays completely upon the programmer the responsibility of choosing the proper algorithm and of implementing it. We think that declarative languages are most suited to be used on parallel architectures because they define explicitly the logic of the program, leaving the control implicit. The “only” worry of the programmer is to describe what the problem really is and not the operation sequence needed to solve it. It is up to compiler and executor to divide the job into different processes and to distribute both them and data on machine processors. It is also for this reason that the number of logic and functional languages implemented on parallel architecture is growing. However most of them only exploit either data-parallelism or process-parallelism. We have chosen to implement a logical language based on the set data structure [Succi81], the Subset Equational Language ([Jay90a] and [JN88]) (called SEL) because it is also suited to be implemented both on a data parallel machine and on a process-parallel one. This paper describes the implementation of the language on a SIMD architecture: the Connection Machine 2 [Hillis].

2 SEL and the Implicit Parallelism

The Subset Equational Language was developed by Jayaraman et al. [JN88] at UNC/Chapel Hill at SUNY/ Buffalo. The exhaustive description of SEL can be found in [Succi81]. In this paper we introduce only the main features of the language that are useful to perform the SIMD implementation (the set data structure definition).

The syntax of subset assertions is simple; it is, however, interesting to observe the syntax by which a set can be defined:

- \{a,b,c\}: this declaration defines a set that is combined by the ground terms specified into the two parenthesis. A set can also contain functors, lists and other sets.

- \{X[\_]\}: this definition can only appear at the left side of an assertion because it specifies a set matching operation. Its behaviour is what we call implicit iteration, i.e., selecting the elements of a set one at a time.

- \{X|T\}: its declarative meaning is similar to the one of \{X\[\_]\}. However, in this case depending on the \{X|T\} definition produces a multiple binding not only for the variable X, but also for the variable T. T is bound to the sets containing all the elements of the starting set but the one referenced by X.

An example of SEL code using subset assertions is shown by the following declarations, where it is defined the permutation of the elements belonging to a set.
perms(\{\}) = \{\{\}\}.
perms(\{X1\}) contains distr(X \ x \ perms(T)).
distr(X \ x \ T1) contains \{\{X1\}\}.

For example, the second argument of the assertion \texttt{distr} specifies an implicit iteration on the incoming set. The execution of the assertion is repeated as many times as the dimension of the set is. At each execution the \texttt{T} variable is bound to a different set element. When the \texttt{perms} assertion is called to perform the permutation on a non-empty set (e.g. \texttt{\{a,b,c\}}), the \{X1\} matching definition causes a multiple execution of the assertion. During each execution the \texttt{X} and \texttt{T} variable are bound to the following ground terms:

- \texttt{X = a, T = \{b,c\}}
- \texttt{X = b, T = \{a,c\}}
- \texttt{X = c, T = \{b,a\}}

If the computing model of the subset assertions is examined, their intrinsic data parallelism is evident. When the square assertion is executed, the square operation may be applied to all the elements of the set at the same time. Obviously, the Subset Equational Language allows writing more complex assertions [MTH93].

However, sets are constituted by different kinds of data, not only simple numerical values and literals, but also structured data, e.g. functors, lists and sets. Therefore, it must be possible to discriminate among the elements belonging to a set according to their values and their formats. For example, an assertion can process only the objects that are functors having identifier \texttt{func2} and second argument \texttt{X}. The set definition that specifies this kind of behaviour is: \{\texttt{func(X,a)}\}.

If we consider the ground set
\{\texttt{func(b,a), func(func(a,a),a), func(a,b), func2(b,a)}\}
the matching operation will produce the following result:

- \texttt{X = b}
- \texttt{X = func(a,a)}

When the \texttt{SEL} abstract machine performs a computation on a sequential machine (e.g. our sequential implementation of \texttt{SEL} has been developed on a SUN4) the right side of the assertions is processed as many times as the number of possible multiple matchings is. In the case of the execution on the Connection Machine, the matching operations are performed by means of a number of actions that is independent from the dimension of the set, i.e. the matching is verified at the same time on all the elements belonging to the set. In the assertion \texttt{as} \{\texttt{func(X,a)}\} \ contains \{\texttt{f(X)}\}, the set definition \{\texttt{func(X,a)}\} declares that the \texttt{X} variable must be bound to the first argument of the functors that have \texttt{func2} as identifier and \texttt{a} as second argument. The \texttt{SEL} abstract machine, therefore, identifies parallelly all data that satisfy the conditions and binds to the \texttt{X} variable not a single value (as in the case of the sequential implementation), but a collection of ground terms (these are the ground terms corresponding to the first argument of the functors having \texttt{func2} as identifier and \texttt{a} as second argument). The assertion \texttt{f} is applied to the new data collection that is referred by the \texttt{X} variable. It is important to notice that also the \texttt{f} assertion can be potentially performed parallelly.

The subset assertions make the \texttt{SEL} language suited to be implemented on a data parallel architecture. In fact, both the operations that are applied to the elements of a set, and the pattern matching have an high degree of implicit parallelism. The Connection Machine 2 is well suited to exploit the features of \texttt{SEL}. The idea that we have used to design the \texttt{SEL} abstract machine consists of distributing the elements belonging to sets on the processors of the CM2. In this way, the Connection Machine becomes like an active memory [MTH93], where the memory cells don't store simply objects, but also can process them.

The principle that we use to exploit parallelism is simple, however we had to face some more involved problems. When an assertion must be applied to a couple of sets, the parallel execution can be performed only on one set, while the execution on the other set must be serialized. The same situation happens when a subset assertion uses another subset assertion. It is again important to have a mechanism that serializes computation on the elements of the Active Memory; this mechanism uses the CM Data Array and the parallel environments (these will be explained later).

3 The Subset Abstract Machine

The implementation of the Subset Equational Language has been performed splitting the design into two different parts. The first part is a compiler that translates the \texttt{SEL} code into a \texttt{SEL} assembly code (called \texttt{SAL}) [Nal88]. The second part is the \texttt{SEL} abstract machine, the \texttt{SAM}, that reads the \texttt{SAL} instructions from a code area and executes them on the physical processor (in the case of the CM2 implementation, the \texttt{SAM} executes the instructions not only on the processor of the front-end, but also on the SIMD processors of the CM2). The \texttt{SAM} belongs to the \texttt{WAM} [AK90] family, since its general structure resembles that of the \texttt{WAM}. However there is an important difference: while the core of the \texttt{WAM} is the handling of unification capabilities, the \texttt{SAM} core is the handling of sets. For this reason some \texttt{WAM} structures are useless and new memory areas are needed.

The Subset Abstract Machine [SM92] is a software layer that models the machine architecture to hide the implementation details of the parallel computer. An outline of the abstract machine is represented in figure 1. The programmer writes a source code that is a collection of equation-
which the same operations are applied. The abstract machine distributes data belonging to sets into the CM2 processors, synchronizes parallel executions and handles the data communications through inter-processor links.

The SAM is made by different modules, each one with a particularly task. In this section, we describe the main elements of the Subset Abstract Machine and underline their behaviour when a matching operation occurs. In this paper some parts of the Subset Abstract Machine (the registers, the code area, the emulator, the heap and the stack) are not discussed, they are described deeply in [SM91b]. The analysis is fixed on the structures that are useful to perform a data parallel processing of sets and on the mechanisms that have been implemented.

4 The SAM and the Parallel Pattern Matching

The parts of the SAM that are involved in the operations of parallel pattern matching are mainly three: the active memory, the CM-Data-Array and the Push_Down_List. Another element is often necessary, it is the parallel environment.

The Active Memory

The active memory (AM) is a collection of memory cells that have a computing power, i.e. they not only store data, but also perform a data parallel execution on them. Therefore the active memory is implemented on the CM2 where a physical parallel execution is possible (while other machine modules, e.g. the stack, the heap and so on, resides on the front-end because their executing model is intrinsically sequential). Every single set element is bound to a specific active memory cell that has a computing capability. Data are stored into local registers and, if they are structured data, into a local heap. Figure 2 shows the structure of an active cell. It is also represented how a structured data is stored into the cells. Register 0 identifies a function (the tag value: STR) and addresses the heap cells that contains the function identifier (func) and the argument number. Because the argument number is 2, then the next two heap cells are used to store its two arguments (the first is an integer and the second a literal). A parallel matching operation must be performed not only on the active cell registers, but especially on the heap of every active cell that contains an element belonging to the processed set. If the func(x,y) definition must be matched with the element of the figure, the Subset Abstract Machine checks as first what active cells contain a function (register 0) and, after, it reads the heap to verify the function identifier, the argument number and the value corresponding to the second argument. Obviously only a subset satisfies the matching; the element represented by figure 2 belongs to this subset. It is important to notice that, where the matching was successful, the first argument of the function must be bound to the variable x. In the sequential implementation, this operation was trivial: the value is copied into the environment register corresponding to \( x \). However, in the data parallel implementation, the \( x \) variable must be bound temporary to a collection of ground terms to allow a data parallel computation. In general, the operations on a set produce some intermediate data on which a data parallel execution is possible. Not only the pattern matching operations, but also the execution of assertions that are applied to set elements. To have a better understanding of this problem, consider the following assertions, where doubleIncrSet increments the elements of the argument set by 2.

\[
\text{doubleIncrSet}(\{x\}) \rightarrow \{\text{incr}(\text{incr}(x))\}, \\
\text{incr}(x) = x + 1.
\]

The innermost call to the incr assertion produces a new collection of data that don't belong to the final set, because they are only transitory data. The outermost call to incr will generate the resulting set. In the same way, the matching operations force the binding of the \( x \) variable with a group of elements that are data used by successive computations. As a matter of fact the elements distributed into the active cells may be separated in two classes:

(a) those generated as intermediate results of a computation,

(b) those produced as a final result of an assertion (resulting set).

The problem is: where intermediate and final results must be stored? There are two different solutions. The first is migrating new object towards free active cells. The second is to save these data into the same cells where the processing has been executed. We have chosen to migrate the elements belonging to a new set, while intermediate results (e.g. the collection of data that are bound to the \( x \) variable) are stored into cells from which they have origin. Such a choice yields an efficient and fast way of generating intermediate sets, which are needed only locally and not globally, therefore they do not need to be burdened by communications. Furthermore this choice keeps a simple structure for global sets: their elements maintain the one to one correspondence with an AM cell, possibly a virtual one, making easy the task of bookkeeping the system, identifying global sets, dereferencing their elements and so on.

The implementation of the AM on the Connection Machine (CM-AM) is built upon two major entities:

1. the Active Memory itself, which contains the sets elements in its cells,
2. the CM-Data-Array, which is used to identify the various sets in the Active Memory (the CM-Data-Array will be analyzed in the next section).

Figure 2: An overview of an Active Cell

595
Addressing Active Cells: the CM-Data-Array

In the last section the structure of the Active Memory has been introduced, but it is also necessary to identify some mechanisms that solve the following problems. The Active Memory is a cloud of active cells: how is it possible to locate only those that store the elements belonging to a particular set? Furthermore, the matching operations enable a part of a set (the part that verifies the matching), possibly creating new references: how can a subcollection of active cells be distinguished among the cells of a set? A new memory area is used to solve the above problems; it is the CM-Data-Array (figure 3). The CM-Data-Array is the interface that is used by the front-end to access the Active Memory. Every entry of it provides all the informations that are needed to process a set in the Active Memory.

- **TYPE**: it specifies whether a set is a permanent set, PER, or an intermediate result, DDT.
- **Tag**: it is used to identify the active cells that are bound to a particular set. A cell belongs to a set identified by a CMDA entry when the value of the tag field is equal to the value stored into a particular register placed in every active cell, the tag register.
- **Register Number**: It determines in which register of the cells the element of an intermediate set are stored; in the case of permanent set, it is the register number 0.
- **Sel. Reg.**: it specifies what is the number of the register that is used to enable or disable a cell during the execution of the pattern matching.
- **Sel. Value**: this is the value that identifies what elements of a set satisfy the matching (in the case of pattern matching processing) or what elements of a set have been processed (in the case of serial iteration instead of parallel execution). The active cells, where the value that is stored into the distributed register identified by the selecting register is equal or greater to the selecting value, satisfy the matching or must be processed.
- **Sel. Elem.**: the value of this field addresses the active cell that is active during a sequential iteration on the elements of a set.

At this point, we have described the Active Memory (the memory area that is used to store the elements belonging to sets) and the CM-Data-Array (the interface between the front-end and the active memory). The next sections explain the mechanisms that allow a sequential computation on sets and the parallel pattern matching execution.

The Parallel Environment

The Connection Machine is a SIMD machine, therefore only one instruction at a time can be executed on all the processors. This peculiarity needs some special care, since there are data parallel flows requiring SIMD structures. If a single operation must be performed on two sets whose elements are allocated in different active cells, the execution may be parallel only on one set while a sequential execution must be performed on the other. A mechanism, called parallel environment, has been devised in order to take care of it. The next assertion presents a situation where such parallel environment is used.

oneProducts(I,{Y1...}) contains (X,Y).
allProducts(I,{Y1...}) contains oneProduct(I,S).

The allProduct execution cannot be parallel, because a nested parallel assertion exists (the oneProduct assertion). Therefore, it is necessary serializing the data parallelism on the first set and exploiting it only on the second set.

When the Subset Abstract Machine must process the set that corresponds to the first argument of the allProducts assertion, it takes a new entry of the CM-Data-Array and saves into it the index of the set and the register number of cells where the values are stored. The tag field is set to the TMP value. In this way data belonging to the set are transformed into temporary data. Afterwards, it writes into a register (located on the front-end), called Active Set Register (AM), the entry number of the CM-Data-Array. The value of ASR identifies the set on which a data-parallel computation is active. The allProducts assertion calls the oneProduct assertion that performs another parallel operation on a second and different set. Before starting a second data parallel execution, the SAM checks the value of the Active Set Register. It addresses an entry of the CMDA, therefore, a data-parallel operation is running. The abstract machine "stops" the current parallel execution and changes the value of the Active Register, the new value corresponds to.

![Figure 3: The CM-Data-Array](image1)

![Figure 4: The CM-Data-Array during a sequential execution](image2)
to the entry of the CM-Data-Array that addresses the new set to process. Besides, it must execute a sequential iteration on the elements belonging to the first set. For this reason, the SAM writes the Selecting Register, the Selecting Value and the Selecting Element of the CMDA entry of the first set. Figure 4 shows the state of the CM-Data-Array and of the Active Memory at this point of the computation (allProducts is applied to the sets: \( \{3, 6, 8\}, \{5, 9\} \)). The content of the Selecting Element addresses the active cell having index 0, i.e. this is the element that has been chosen and bound to the \( X \) variable of the assertion allProducts.

The other register of the same entry identify the elements of the set that have not been processed. However, the only CM-Data-Array is not sufficient to handle nested data parallel computations. In fact, when the data parallel operation of the oneProduct assertion is finished, the previous data-parallel processing must restart; it is necessary to save informations about the parallel operations that are stopped.

The SAM uses the Parallel Environment. Like the normal environment, it is saved onto the stack. It contains the old value of the active set register (in this way the previous set that was processed parallelly is known), the old value of the selecting value field (in fact it is possible that other computations can change its value) and the pointer to the previous parallel environment. When the processing of the first data parallel operation of the oneProduct assertion is finished, the parallel environment is used to restore the first set as the active set. At this point the operations are repeated. However, when the second set of the oneProduct assertion must be processed, the sequential iteration is applied only to the elements that have not been processed. Another use is chosen, the value into its selecting register is decremented and, at last, it is bound to the \( X \) variable.

**The Push\_Down\_List**

A situation that merits a particular analysis is the computation of two nested sets that require pattern matching operations. They don't set on the entire set, but only on a subset of it. We consider the following trivial SEL program:

1. `allProducts2((func1(X),),S)` contains `oneProduct2(X,S)`
2. `oneProduct2(X, (func2(Y)_1))` contains `X*Y`.

The figure 5 outlines the execution flow of the assertion.

It is similar to the processing of two nested sets that has been discussed in the previous section, but, in this case, the multiplication is applied only to the elements that satisfy the matching. In the subset abstract machine we use the Push\_Down\_List to perform a data parallel pattern matching operation. When the emulator processes the SAL code corresponding to the head of the allProduct2 assertion, it saves into the PDL a template of the pattern to be matched. Figure 6 shows a typical template that is stored in the Push\_Down\_List: the first cell identifies the structured data as a function, the second cell specifies that is a function having identifier func1 and only one argument, the meaning of the first cell will be explained later. When a data parallel pattern matching occurs, the SAM executes the following operations:

1. a new register into every active cell that contains an element of the set is allocated to store the selecting value and is initialized to zero,
2. the selecting register and the selecting value fields of the CM-Data-Array entry that corresponds to the set to be processed are initialized: the selecting register field addresses the register allocated in the previous operation step, the selecting value field is set to 0 (at this point every cell is active).

3. The objects that are saved into the Push\_Down\_List are broadcasted to the active cells addressed by the CMDA entry and the match is verified. In the example, at first, it is broadcasted the (STR,1) couple and every cells checks if the element that is saved into itself is a function. If the match fails, the value of the selecting register is decremented (in this way the cell is disabled). The next couple to be sent is (func1) and, therefore, the elements of the set that are not functions with identifier func1 and only one argument are discarded. At last, the couple (VAR 120) is broadcasted.

The couple (VAR 120) is very important, because it addresses a Y register of the environment (see fig. 6). This register, at the end of the matching, will contain a reference to a distributed data. For this reason, when the Subset Abstract Machine stores in the PDL the (VAR 120) object, it also allocates both a new entry in the CM-Data-Array and a new register into every active cell (only those that contain the elements of the set). At the end of the matching of Set1 (see figure 6) a new temporary distributed data has been built using a single data parallel operation. The same pattern matching processing is applied to Set2 producing a second temporary distributed data. At this point, the computation can continue as in the previous section.

### 5 Conclusions

Several are the issues still open in this research. Abstract analysts and garbage collectors are under study and development. A different way of mapping the Active Memory on the hypercube by means of hash tables is almost completed. Network optimizations and compilation enhancements are also hot points.
Acknowledgments

This work has been partly supported by the Italian Ministry of University and Scientific Research (40% funding). Most of the ideas presented in this paper has been discussed with B. Jayaraman. The authors thank researchers of DII/Parma for their support in the usage of the Connection Machine.

References


