Yale University
Department of Computer Science

Footprints of Dependency: Towards Dynamic Memory Management For Massively Parallel Architectures

Marina C. Chen, Michel Jacquemin

YALEU/DCS/TR-593
January 1988

This work has been supported in part by the Office of Naval Research under Contract N00014-86-K-0564. Approved for public release: distribution is unlimited.

Footprints of Dependency: Towards Dynamic Memory Management for Massively Parallel Architectures

Marina C. Chen Michel Jacquemin

Computer Science Department, Yale University

Abstract

Dynamic management of memory hierarchy is a cost-effective means to achieve performance on conventional machines as well as vector processors. On parallel machines, fragmentation of memory among processors is a new issue which must be addressed together with the issue of memory hierarchy.

In this paper, we present a dynamic memory management scheme for massively parallel architectures. "Footprints" of dependency among data elements are recorded and on-the-fly partial sorting of data elements according to the footprints are performed. As a result of reorganization of data, there is a significant increase in speedup over unmanaged data. The scheme has been implemented on the Connection Machine.

1 Introduction

It is generally the case that there will never be enough processors to allow data elements of very big problems to be allocated one per processor; even with massively parallel architectures that have as many processors as 10^5 and up. Processors will have to be shared in some way between several "tasks". In other words, a virtual processor system must be in place. This raises an important question that will be the subject of the text to follow: how shall the data associated with these tasks be arranged among different processors and memories in order to optimize performance of execution? First, we must take a look at why performance needs to be optimized.

Due to dependencies within the algorithm's computations and the possibility of unpredictable, irregular distribution of data; un-managed data may cause load imbalance, extra cache or page misses, and/or extra communications between processors. Amount of dependencies means the number of intermediate computations that are strictly necessary to be able to compute the result of the algorithm. The amount of dependencies can be thought of as an inverse measure of intrinsic parallelism of the algorithm.

How does dependency affect parallel program execution? Suppose a computation has a data-dependency graph as shown in Figure 1(a). If nodes are distributed to processors' local memory as described in 1(b), then there is a serious imbalance between the two processors. Nodes 2 and 3 of the graph can be executed at the same time, but they reside in the same processor. The same goes for nodes 4 and 5. However, if the nodes are distributed as showed in 1(c), parallelism of the graph will be fully exploited. Another example is when a computation has a data-dependency graph that is much larger than the number of processors. Good organization of the data for each processor should allow nodes that are close together in the graph to be in contiguous locations in memory (or on disk) so that they can be paged-in to the cache (or main memory) together.

Memory hierarchy is a cost-effective means to providing sequential machines with very large memory spaces [12]. Trying to achieve this involves keeping the effective memory access time close to the time provided by the fastest (and smallest) memory level of the system. Some usual levels encountered, (which are ordered by increasing access time) are: registers; one or two level caches; physical memory; and virtual memory disk system. Achievement of this relies on some assumptions on the way processes execute and access data. The main assumption is locality, which actually can be broken down into two types: locality in code (of control), and locality in data. At times, locality in code says that the proportion of jump instructions in the program are reasonably small, and most instructions are executed in sequence. At other times, it assumes the programs are mostly built of loops that repeatedly execute the same code. Locality in data says, that at any given instant, the probability of the program accessing data close in memory to the data it has accessed in the last few steps is much higher than data that is far in memory. Thus, locality plays a central role in making an efficient use of the memory hierarchy of sequential machines.

Parallel machines also have memory hierarchy. Shared

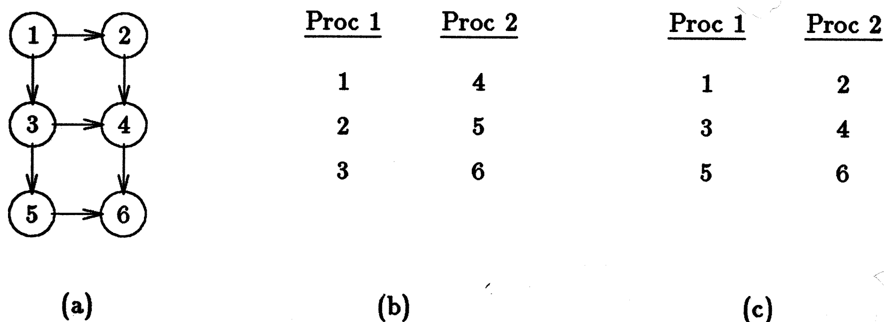

Figure 1: Example of a data-dependency graph and allocation of nodes to processors

memory machines have caches and/or local memories in each processor; and message-passing machines make the distinction between the processor's own memory and the memory of the other processors. Massively parallel machines need to have parallel disk systems in order to be used efficiently without suffering from the I/O bottleneck at the host. Furthermore, memory in a parallel machine is fragmented among all processors.

Let us look at code and data distribution problems in parallel machines. Code must be distributed among the processors. We can distinguish two cases in code distribution. In the first case, there is a single identical program for all processors (SPMD, etc., see [8]), residing in whole in their memory. Processors execute a (perhaps different) subset of the instructions through conditional control. This situation being very similar to the uni-processor case, where processors can use instruction buffers or caches. The SIMD machines are a special case of this, where the program resides in the "front-end" and is distributed to all the processors. In the second case, each processor has a different program to execute and the number of such programs is large. The problem of their distribution among processors is similar to the distribution of data among processors; dependency of control replaces than dependency of data.

Thus, without loss of generality, we can focus on the distribution of the data. The role of locality in the context of the parallel machine is related to both dependency and *anti-dependency* of data. Anti-dependency manifests itself as the spread of the data to processors: if two data items are not inter-dependent, they should be be put on two different processors as much as possible to allow more parallelism. Dependency manifests itself as the spread of data in time within a given processor: data should be organized in such a way that a processor would have to access data mostly from what it has previously computed.

To take advantage of this dependency and anti-dependency of data in dynamic memory management, a stronger requirement is needed by the "operating system". The operating system must store the dependency information. Namely, the "footprints" of dependency.

2 Description of the Problem

We can classify computational problems into five levels of difficulty according to dependency and regularity. Regularity is a useful property which can give rise to more efficient algorithms and simpler data representations (e.g. systolic algorithms), rather than non-regular problems. The first class of problems has few or no dependencies and high regularity in its data structure i.e., the problem of matrix multiplication. These are algorithms that have "embarrassingly high" parallelism. The second class of algorithms consists of those with significant dependencies among inner computations, but still a very high regularity, like the computation of the LU-decomposition. The third class includes algorithms which have few dependencies and little or no structure, like the sparse matrix-vector product. These three classes can be qualified as relatively easy to parallelize because there are few dependencies, and the organization of data among the processors is greatly eased and not critical (in terms of minimizing the communication traffic). Also, regularity allows the ability to foresee communication patterns and work loads at compile time.

The fourth class of problems are those that have significant amount of dependencies and little or no known structure. This is the class of problems for which we are interested in addressing memory management issues. We'll restrict ourselves to the static dependency graph that does not change in the course of computation. There is a fifth class of problems that is even more difficult. These problems have a lot of dependencies, whose dependency structure is dynamic, constantly changing with time. The sparse LU-decomposition problem falls in this class. Here, the dependency structure is being modified by the "fill-in" during back-substitution. For this kind of problem, rearranging the data statically will be of little help. On the other hand, the cost of dynamic memory management will be too expensive to justify because there is no repetitive use of the same dependency graph over which the cost can be amortized. The usual solution for this class of problems is to design the algorithm differently if possible (e.g. dissection)

or to use hashing to distribute data (universal hashing functions).

2.1 Representing Computation by DAG

Any computation can be represented by its dependency graph, whose nodes represent the computations performed by the algorithm and whose edges represent the communications between nodes ([9,10]). By the computations between nodes we mean the dependencies of a node's computations upon other nodes' outputs. This dependency graph is a directed acyclic graph (DAG). Such DAG has been widely used in compiler flow analysis (see, for example, [5]).

We assume here that the computation is described in such a way that each node of its DAG represents computation of appropriate granularity with respect to the target parallel machine. The issue of controlling the granularity of a computation is an interesting one but beyond the scope of this paper. (If you are interested in reading further information on this subject, see [4] for compile-time method and [11] for run-time method of granularity control).

2.2 Characteristics of DAG

What is the most economical, and the fastest way to execute this computation on a parallel machine, with as many processors as we want? The model we assume will be described precisely later. For now, let's consider that communications are the dominating costs and the computations performed inside the nodes are negligible compared with the communications. It is easy to see that the fastest way to achieve this is proportional to the *depth* of the graph, in other words, the longest path from an initial node — one whose computation doesn't depend on other nodes' computation, to any other node. Each node in the graph can perform its own computation only when all of its immediate predecessors have finished computing their values. Notice that this node won't have to execute any time afterwards.

Thus, an "optimum" way to perform the computation of the DAG is to execute all the nodes that have no predecessors during the first step. Then, repeatedly execute all the nodes that just had all their predecessors' values determined.

Some interesting characteristics of a dependency graph are its depth and its width. The above-mentioned depth of the graph measures the lower bound of time that can be achieved on a parallel machine. The *width* of a particular graph is defined to be the number of nodes that are at the same depth in the graph. We are interested in both the maximum width and the average width. The maximum width tells us how many processors are necessary to achieve this optimum time (regardless of allocation problems), and average width gives a rough measure of what the average parallelism of the algorithm is, and what the "cost-effective" number of processors to allocate is.

In general, with very large problem sizes, the situation looks as follows: we have a very large set of data, far outnumbering the processors of the parallel machine. At every time step in the "optimal" execution, there might be a limited portion of the nodes which are able to be executed due to dependency. By limited, we mean something significantly smaller than the number of nodes n, something that's not $O(n)$ but $O(n^\alpha)$, with $0 < \alpha < 1$, for example $O(\sqrt{n})$. For a machine without proper memory management, there can be a significant waste of processor cycles since the active nodes are limited. For a fixed number of processors, computation will slow down linearly as the problem size (total number of nodes) grows. What is important here is that the goal of the memory management system is to enable the parallel machine not to slow down when the problem size grows, as long as the number of active nodes — which is much smaller than the problem size, are smaller than the number of processors.

2.3 DAG-based Memory Organization

The fact that there is much more data rather than processors implies that a lot of data elements are packed into each processor. If the structure of the dependency graph is arbitrary and unknown at the beginning, we can do very little else than looping through all the data at each step, in order to find out which nodes in each processor are ready to "fire". This can't be avoided the first time we perform the computation, however, we can do much better the next time we use this same dependency graph.

Suppose that the computation governed by the same dependency graph is going to repeat a lot, then the cost of performing the first iteration without dependency information can be amortized over many iterations. This assumption is often valid, such as in the case of solving a linear system of equations where one wants to iteratively solve many systems with the same matrix — which determines the dependency graph, and different right-hand sides. Thus, given that the dependency graph is the same over time, nodes in the graph will be able to fire at exactly the same step in every iteration. Information about the structure of the graph can be gathered during the first iteration with little overhead. Computation can then be reorganized thanks to this information before the second execution.

In summary, the problem we consider is one of repeated execution of different computations that have the same dependency graph. Since the amount of data (or the number of nodes) are much larger than the number of available processors, the data needs to be reorganized in a way that enables fast execution despite the fragmentation and the hierarchy of memory that is present in the parallel machine.

3 An Example

We choose sparse matrix forward solve as an example to illustrate our data reorganization method. Even though its data dependency graph is a special case of all possible DAG's but it has all the "richness" of DAG's in terms of possible structures: any directed acyclic graph can have its adjacency matrix brought to a lower triangular form by some permutation of the nodes. The sparse matrix forward solve has the nice property that the computations performed in it are all similar and quite simple: it enables us to focus on the effects of rearrangements of the data on performance.

Thus without loss of generality, we describe our implementation of the method in terms of the sparse matrix forward solve. Given a sparse lower triangular matrix A and a vector y, we want to solve the following system for x,

$$Ax = y$$

where A is the $n \times n$ sparse matrix, whose zero-structure is arbitrary and unknown at the beginning of the algorithm.

The execution of forward solve on a sequential computer would often solve for the first variable x_1, then using this value, solve for x_2, and so on. In a parallel implementation we are interested in getting the highest speed-up. When the matrix is not sparse, solving for any x_i would require to know the values of all the preceding values $x_1 \ldots x_{i-1}$, so we are forced to compute them in order and get very little parallelism from this. In the sparse case, due to the zero elements in the matrix, some "dependencies" disappear, relaxing the constraint of sequentiality between the x_i's. This gives rise to more parallelism. Unfortunately, when the zero-structure of the matrix is unknown, we can't predict how this parallelism is going to come out in a precise way. And, of course, there are still a lot of dependencies. The structure of this forward solve can be viewed as a dependency graph, where the nodes represent the variables x_i and the edges representing the non-zero elements of A. The zero-structure of the matrix gives us the adjacency matrix of the dependency graph. In other words, $A[i,j] \neq 0 \Leftrightarrow$ the value of x_j is needed to compute the value of x_i. Since the matrix is lower triangular, it defines a directed acyclic graph.

3.1 Representation of a Sparse Matrix on the Connection Machine

We present the example with a Connection Machine [6,7] implementation, where the effect of the memory fragmentation and hierarchy comes at the smallest scale: one matrix element is assigned to each processor, and there is penalty in the memory access time if all the memory references performed by processors are not to the very same location in their perspective local memory.

We allocate one processor per variable x_i and one per non-zero matrix element of row i. The processors for one given row are allocated contiguously in a segment following the processor containing the associated variable (to take advantage of fast parallel prefix computation). This means that the processors containing the $A[i,j]$'s will be set up in a contiguous segment following the processor containing $A[i,i]$, y_i and (when it is known) x_i (figure 2). From now on, we will refer to the processor allocated to x_i as the node processor i and to the processors assigned to $A[i,j]$ as the edge processors (i,j), by analogy to the dependency graph.

We organize all the data elements this way, until we run out of processors. Then we start back from the first processor, creating a new layer of data, and so on. This initial organization has the advantage of spreading data evenly among the processors.

A typical computation step is (for a fixed i):

- each of the edge processors (representing $A[i,j]$) gets the value of x_j (assuming it is already computed), and multiplies it by its $A[i,j]$ value
- using parallel prefix on the segment we compute the sum of the products, subtract y_i (right-hand-side) from it, and divide it by $A[i,i]$ leaving the result in the node processor.

3.2 Keeping Footprints of Dependency

At this initial state, there is no sufficient knowledge of the dependencies between the data elements. Hence, processors have to pick out the active data elements from their memory for processing. To find out whether a given processor should be active at each time step, each processor loops through all the layers of data in its memory. By checking the availability of all the values necessary for the computation of a particular variable x_i for which it is responsible, the processor knows whether it should be active at that time step or not. Each processor along the way of the above computation records for every data element (variable x_i), the step at which it was active (i.e. when it performed its computation). This information will be used to perform the reorganization of the data. It is done by using a counter to keep track of the step number in each processor. A "time-stamp" (the depth of the data element in the dependency graph) is attached to the data elements when they are active. This extra record-keeping step is an inexpensive operation on almost any machine. The collection of time stamps now contain enough information to derive both the dependency and anti-dependency relations among the data elements.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 4 & 5 & 0 & 1 \end{pmatrix}$$

Proc	1	2	3	4	5	6	7	8
A	1	1	2	1	3	1	4	5

Figure 2: How a row of the sparse matrix is represented

```
forall i in [1..n] do known[i] := false;
iteration := 1;
repeat
  forall i in [1..n] do
    if not(known[i])
      then if AND(known[j] | j in indices(a[i]))
        then {
          x[i] := (y[i] - SUM(a[i,j]*x[j] | j in indices(a[i]))) / a[i,i];
          known[i] := true;
          timestamp[i] := iteration; }
  iteration := iteration+1;
until AND(known[i] | i in [1..n]);
nb_iterations := iteration-1;
```

Figure 3: algorithm used before data reorganization

3.3 Data Reorganization according to Footprints

In any subsequent iteration of solving $Ax' = y'$, the footprints of dependency (and anti-dependency) can be used to reorganize the way data is stored, thus increasing the performance. The idea of the reorganization is very simple: just sort[1] the variables x_i's by their time stamps so that those x_i's with the same step number will be in contiguous places in the same layer, or in adjacent layers (to be precise, actually the whole row of matrix elements $A[i,j]$ are moved together with x_i as one block once the sorting is done).

Having done that, the execution can now be greatly sped up. At each time step of the computation, there is no need to loop through all the layers. It is necessary to loop only through those limited number of layers that are stamped with that particular time step. Exactly how many layers for each time step relates to the "width" of parallelism in the algorithm, and the number of processors available. If the number of processors used is significantly higher than the average width, then a large majority of steps will need data in one layer so the performance will be close to the optimum. Degradation in performance due to the sequentialization starts to occur as the number of processors become close to the order of the average width. Some experimental results of these cases on the Connection Machine are presented in the next subsection.

The algorithms used before the reorganization is described in figure 3 and the one used after the reorganization is given in figure 4. The lower triangular, $n \times n$, sparse matrix **a** is given and stored as described earlier. The reader can notice that the x's will be computed in exactly the same order in both algorithms.

The reorganization procedure is described in figure 5.

3.4 Experimental Results

We implemented the idea described in the previous section on an 8K processors Connection Machine using the sparse matrix forward solve problem for matrices with up to 10^5 non-zero elements. we varied the problem sizes, the number of processor used, the number of layers, and the amount of parallelism in the problem (by acting on the "sparsity" of the matrix).

Figure 6 represents the speedup of the parallel execution with rearranged data over the parallel execution with the initial un-organized data. Given a fixed problem size (with fixed depth), we implemented our method with increasing number of layers (by decreasing the number of processor

[1] The Connection Machine has a very fast sorting primitive. In general, we don't need to do a sort of the complete data set at once, it is enough to perform a partial sort "on the fly"

```
for iteration:=1 to nb_iterations do
  forall i in [1..n] do
    if timestamp[i]=iteration
      then
        x[i] := (y[i] - SUM(a[i,j]*x[j] | j in indices(a[i]))) / a[i,i];
```

Figure 4: algorithm used after data reorganization

- **rank[i]** := index of **timestamp[i]** in the sorted sequence of **timestamp**'s
- allocate the processors as in section 3.1, but following the order defined by **rank** : first allocate processors for row **i** such that **rank[i]**=1, and so on.

The next table shows what the organization of data elements will be after the reorganization procedure described above. The numbers show the timestamp associated with each data element.

Processor		1	2	3	4	5	6	7	8
Layer	1	1	1	1	1	1	2	2	2
	2	2	2	3	3	3	4	4	4
	3	5	5	5	5	6	6	6	7
	⋮								

Figure 5: reorganization according to timestamps

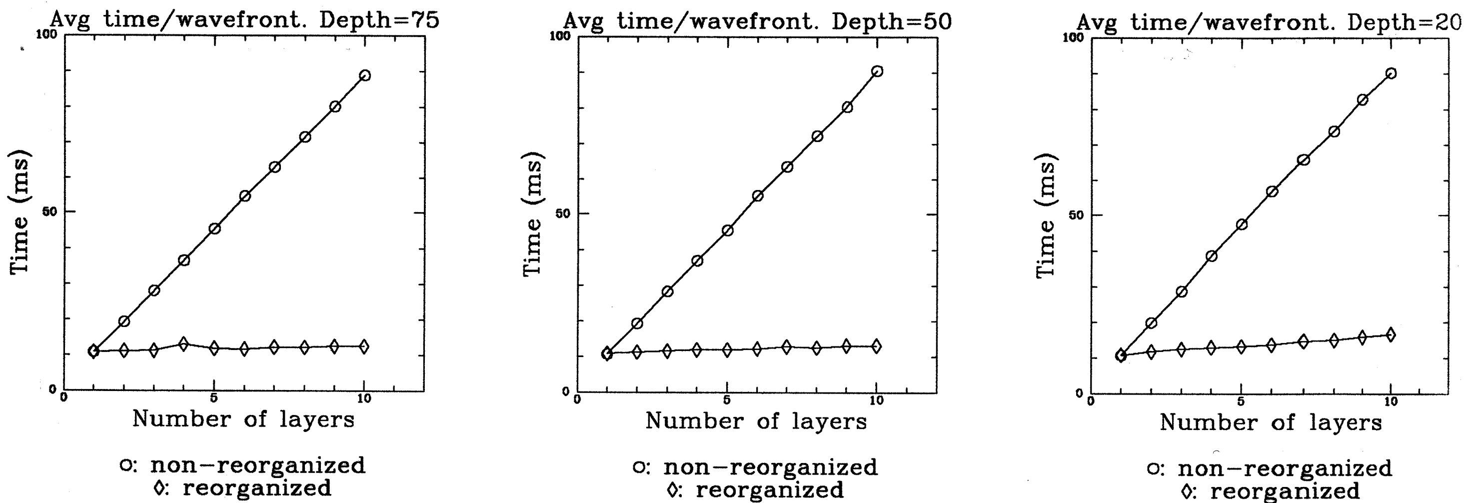

Figure 6: Average time spent per wavefront, for non-reorganized and reorganized data, in three different cases.

used). We measured the average time per wavefront (a wavefront is the set of nodes that have to execute at the same time, the number of wavefronts in a graph is equal to its depth) taken by both the reorganized version and the un-organized version. Measurements were made for three sets of problems which have different depth in their perspective dependency graph, and some observations about these results are mentioned below.

- The time spent per wavefront in the un-organized case is, for all practical purposes, linear in the number of layer, while in the reorganized case it is almost independent of the number of layers.
- When the number of layers becomes close to the depth of the graph (as in the third graph of figure 6) the time of the reorganized version is growing.
- The overhead due to keeping track of the dependency graph is very small. It only amounts to remember the step at which the processor was active, a local and very cheap operation.
- The overhead due to the reorganization of the data is more significant, but most of the time, it only occupies a small fraction of the total execution time of each iteration on the reorganized DAG. It is not a constant fraction because the execution time for the reorganized DAG is essentially proportional to its depth, while the reorganization's main operation is sorting on the node processors. In all the experiments we have made, this overhead is less than 10%.

In all cases, data reorganization results in significant speedup, almost linear in the number of layers used.

4 Memory Management on SIMD Disk Systems

When the data sets we are considering become very large and exceed the available memory in all the processors, we have to have the data residing on a external storage device. For parallel machines, especially with large number of processors, a single disk system would create a bottleneck in terms of data access. Newer machines like the Connection Machine are associated to a parallel disk system. When the machine is MIMD and each processor has its own disk, the disk management problem for each processor is exactly the same as for single processor machines: they just can have their own virtual memory system working independently of one another. On a machine with an SIMD disk system, the problem is different since once a processor is accessing the disk system, all the processors are penalized. Therefore it is important to have an organization scheme that makes all the processor access the disk at the same time. We believe the scheme described in this paper can be applied to an SIMD disk system and improve the performance of the whole system.

Considering a disk system with a very big data set, the situation would be the following: all the layers of data elements reside on disk and some of them are loaded into the processors' memory. Our method of reorganization (assuming there is an economical way to sort the data elements on the disk) insures first that each layer brought into memory is exploited and won't have to be brought back later. Furthermore, there is locality of access across the layers, insuring that when a block of layers is brought from disk to memory, all the layers will be used one after another and this block won't have to be brought back again from disk.

5 Integration into a Parallel Programming Environment

One of the main goals of a parallel programming environment is to free the user from the burden of mapping the computations to the processors. The parallel programming language *Crystal* [1,2,3] enables the user to specify the computations to perform without having to include the mapping. The dependency graph of the computation can, in most cases, easily be generated from the Crystal code. In some cases, the dependency graph can be generated at compile-time, otherwise it is generated at run-time. Crystal is a very high level language, and is purely functional: expressing a computation in Crystal does not introduce dependencies that are not part of the problem (e.g. due to the use of sequential program constructs such as lists, serial loops, etc).

The reorganization of data can be performed automatically by the underlying system since the same dependency graph is obtained once the program has executed the first iteration

Figure 7 shows a Crystal program expressing the computation in sparse matrix forward solve. The equation exactly defines the dependencies among the `x`'s. Hence, the extraction of the dependency relation is easy to perform: `x(i)` depends on all the `x(j)`'s such that `j` appears as column index in row number `i` (as defined by the function `indices(a(i))`). This dependency can be automatically transformed into code like in figures 3 and 4. The code in figure 3 is an almost direct translation of the Crystal code, except for those instructions added in order to (1) insure the execution order according to dependencies (instructions involving `known[.]`), and (2) keep track of the timestamps (instructions involving `timestamp` and `iteration`). The code in figure 4 is also a direct translation, assuming the timestamps are known (from a previous execution).

```
!!! The input (sparse) matrix, a, represented as a sequence of rows,
!!! each row consisting in a sequence of pairs [value,column_index],
!!! the first pair being associated with the corresponding diagonal
!!! element of the matrix
!!! The right-hand side, y, is a sequence of values
a = [[[1,0]], [[1,1],[2,0]], [[1,2],[3,2]], [[1,3],[4,0],[5,1]]],
y = [1,4,-1,2],
n = ||y||,

x(i) over D =
  ( y(i) - inner_prod(off-diag(a(i)),
                      off-diag([x(j)| 0<=j<i : j in indices(a(i))]))
    / a(i)(0)
  where
  ( D = [0..n-1],
    indices(tuple) = [tuple(i)(1) | 0<=i<||tuple||],
    off-diag(tuple) = [tuple(i) | 1<=i<||tuple||],
    inner_prod(tuple1,tuple2) =
      \+ [tuple1(i)*tuple2(i) | 0<=i<||tuple1||]
  ),
```

Figure 7: sample Crystal program for sparse matrix forward solve; the input matrix is the same as in figure ??

In general a program will contain a large number of equations. Only some of these equations will be defined over domains (using the `over` keyword in Crystal): these are the equations defining the dependencies which the system will use for this type of optimizations.

6 Concluding Remarks

We have described a method of reorganizing data in a parallel machine so that maximum parallelism can be exploited. The method applies to problems that make repeated use of the same dependency graph. The improvement applies to all executions except in the first iteration where the overhead of the reorganization is incurred. In most cases, such overhead is negligible.

The method presented in this paper can be applied to problems for which the dependency graph changes little over iterations. The footprints of dependency gathered in the first execution are going to be updated at each iteration. The question is then when to perform data re-organization. If the dependency graph in the subsequent execution is a sub-graph of the initial one, this data organization becomes overconstraining and therefore sub-optimal. On the other hand, if the number of edges in the dependency graph grow with the computation, the data organization will be sub-optimal due to idle processors. In the case where incremental reorganization of data is inexpensive, optimal execution can be maintained. Otherwise, a trade-off between using out-of-date organization and the cost of re-organization must be made.

The performance behavior of our dependency method is analogous to paging of virtual memory in uniprocessor systems in the following sense: there is a transient period in which the dependency of the problem is "learned", which corresponds to the locality being learned by a conventional virtual memory system. Then comes a "steady state" period during which most memory references "hit" the working set. In our case, the steady state begins upon the system knowing the DAG. From this point on, load balance and good performance in accessing distributed data will be achieved. As soon as the program enters a new dependency structure, a new cycle of transient learning is triggered and a steady state period follows. Similar to virtual memory management for conventional machines, our method for massively parallel machines is effective, simple and robust. Such are systems that have been proven to work well in practice.

References

[1] Marina C. Chen. Crystal: a synthesis approach to programming parallel machines. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, August 26-27 1985.

[2] Marina C. Chen. Transformations of parallel programs in crystal. In *The Proceedings of the IFIP 86, Dublin,*

Ireland, September 1986.

[3] Marina C. Chen. Very-high-level parallel programming in Crystal. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[4] Marina C. Chen, Erik DeBenedictis, Geoffrey Fox, Jingke Li, and David Walker. *Hypercubes are General-Purpose Multiprocessors with High Speedup.* Technical Report 11354-880104-02TM, AT&T Laboratory, 1987.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[6] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

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

[8] Alan H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[9] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M.J. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[10] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), 1986.

[11] Joel H. Saltz and Marina C. Chen. Automated problem mapping: the crystal runtime system. In *Hypercube Multiprocessors*, pages 130–140, SIAM, 1987.

[12] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1987.