(5)

Graph Algorithms for Supercomputers

Martin Nilsson and Hidehiko Tanaka
Hidehiko Tanaka Lab., Dept. of Electrical Engineering,
The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN

Abstract: Algorithms on graphs are often inherently massively parallel, making them attractive for execution by supercomputers. However, current supercomputers are optimized for numeric processing of matrices, and do not appear easily adaptable to manipulation of non-numerical, irregular pointer data structures, such as are used in graph algorithms.

In this paper we present an efficient algorithm which does indeed enable supercomputers to efficiently traverse pointer structures, composed of varisized records. This algorithm can then be used as the key component of various graph algorithms. As an example, we give an algorithm for parallel varisized-cell garbage collection.

Keywords: Supercomputer, Non-numerical, Parallel, Graph, Traversal, Algorithm, Garbage collection.

1 Introduction

Recent years have seen a strong boost of symbolic processing and symbolic programming languages such as Lisp and Prolog, especially in fields related to Artificial Intelligence. At the same time, the development of high-speed supercomputers has been explosive. However, current supercomputers are almost exclusively designed for fast execution of numerical programs, defined on regular and static data structures, such as for solving differential equations on grid patterns.

Many non-numerical problems, e.g. garbage collection and massively parallel unification, are both computationally heavy, and inherently parallel. It is tempting to try to also implement algorithms for such problems on supercomputers. The trouble here is that non-numeric data structures are not as regular as the grid patterns: They are a seemingly arbitrary mess of tangled pointers,

In this paper, we will however describe efficient algorithms for handling pointer structures by SIMD computer architectures, such as vector parallel supercomputers. We will describe a key algorithm in detail, and as an example show the algorithm applied to parallel garbage collection of varisized structures.

The algorithm is applicable for a large class of supercomputers, including Hitachi's S-820 [5] and the Connection Machine of Thinking Machines, Inc. [3].

In general terms, our key problem can be stated as follows: Given a set of nodes of a graph, compute the set of all successor nodes. Nodes may have different degrees, i.e. different numbers of successors.

If all nodes were of the same degree, then a pair of nested loops - one indexing over the nodes, and the other indexing over the successors of each node - trivially solves the problem. But when nodes are of different degree, this solution will not work. If we first collect all nodes of the same degree and process them together, much overhead will be required, if there are nodes of many different degrees. A more efficient method is necessary.

This paper is organized as follows: In the rest of the introduction we will more precisely state the problem and define the terminology and requirements of target computers. In section 2, we will give the algorithm with a detailed example. In section 3, we apply the algorithm to parallel garbage collection. In section 4, we summarize and the results.

1.1 Graph Representation and Terminology

We represent nodes by *records*. Each record consists logically of several items: The *degree* of the record, a pointer to an array of successor pointers, and the array of successor pointers itself.

We use three vectors to represent records: *degree*, *tail*, and *succ*. *degree*[*i*] is the degree of record *i*, *tail*[*i*] is the index into a vector of successors, *succ*, of the first successor of record *i*. The successors of record *i* are thus: *succ*[*tail*[*i*]], *succ*[*tail*[*i*] + 1], ... *succ*[*tail*[*i*] + *degree*[*i*] − 1].

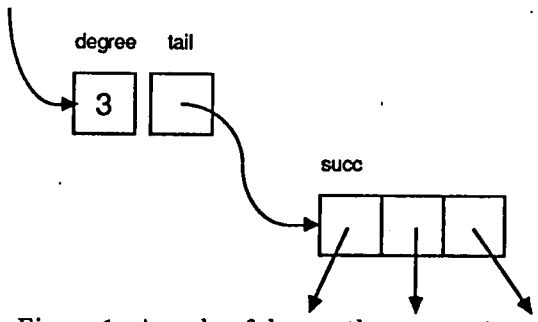

Figure 1: A node of degree three.

We can now formulate the problem as follows:

> Given a set A of N record pointers, represented as an array. Compute the corresponding set B of successor pointers $succ[tail[A[i]] + k]$, for $1 \leq i \leq N$, and $0 \leq k < degree[i]$.

1.2 Computer Requirements

We require that the computer is able to vectorize and execute in parallel simple arithmetic operations [6] such as

```
for i = 1 to N {
  a[i] = b[i] + c[i]
}
```

We will also require two other operations which, such as Cray 1, exist on modern supercomputers such as S-820. These operations are *list vector operations*, and *linear iteration*. These correspond to single machine instructions on both S-820 and the Connection machine.

List vector operations are operations where the index of a vector reference is itself a vector:

```
for i = 1 to N {
  a[b[i]] = c[i]
  d[i] = e[f[i]]
}
```

The linear iteration operation calculates all partial sums of the linear recurrence equation $a[i] = a[i-1] + b[i]$. A typical program for which a vectorizing compiler generates this instruction is

```
for i = 1 to N {
  a[i] = a[i-1] + b[i]
}
```

For the sake of readability, we will use this pseudo-programming language notation in order to describe the algorithms. There should be no problem for the reader to translate the programs into explicit vector operations, if necessary. All programs given are compilable by the S-820 vectorizing compiler.

1.3 Related work

Much work has been done on numerical algorithms for supercomputers, but very little on non-numerical algorithms. This is probably due to the fact that it is less than obvious whether non-numerical processing will at all be efficient at all on current supercomputers. Some work on non-numerical programming languages for supercomputers is [1, 8, 9, 10, 11, 12, 13, 14, 16]. There is some work on SIMD computers which are not generally available, or which are special-purpose [2, 7, 15]. An exception is the Connection Machine [3], which does very well support non-numerical applications, and which also has some very interesting algorithms described for it [4]. Actually, the algorithm we describe in this paper becomes particularly efficient on the Connection Machine, thanks to its parallel prefix operation, which is a generalization of linear iteration.

2 The Algorithm

We will first describe the algorithm by going through a simple example and illustrate the process by figures. Then, we will give the algorithm as a program.

The idea of the algorithm is to compute indices into the vector of successors by linear iteration. In order to do this, we will use three auxiliary vectors, *acclen*, *succdist*, and *succindx*. *acclen*[*i*] gives the position in B of the first successor of node *i* to be stored in B, i.e. *succ*[*tail*[*i*]] will be stored in $B[acclen[i]]$.

Suppose first that we have three records, of 3, 2, and 4 elements, respectively. *acclen* should then be

set to the values 1, $1+3=4$, $1+3+2=6$. We can also use the opportunity to compute $acclen[N+1]-1$, which is the total number of successors, M. Here, $M=3+2+4=9$.

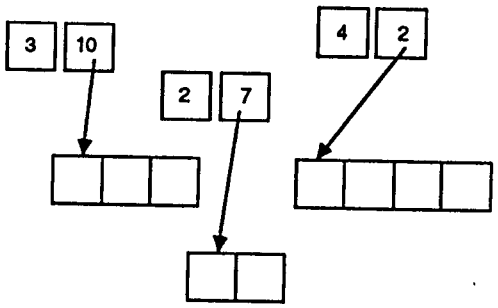

Figure 2: Example records.

acclen is computed by linear iteration in the following way:

$$acclen[i] = \begin{cases} acclen[1] & i = 1 \\ acclen[i-1] + degree[i-1] & 2 \le i \le N+1 \end{cases}$$

succdist is used to compute the indices of all successors. It contains the successive differences between the successor pointers. We first initialize it so that all elements are one. Then we first set $succdist[1] = tail[1]$, and then we modify all elements at positions $acclen[i], 2 \le i \le N-1$, to be $tail[i] - tail[i-1] - degree[i-1] + 1$.

In our example, $succdist[1] = 10$, and $succdist[i] = 1$ for $i = 2,3,5,7,8,9$. The modified elements are $succdist[4] = tail[2] - tail[1] - degree[1]+1 = 7-10-3+1 = -5$, and $succdist[6] = 2-7-2+1 = -6$.

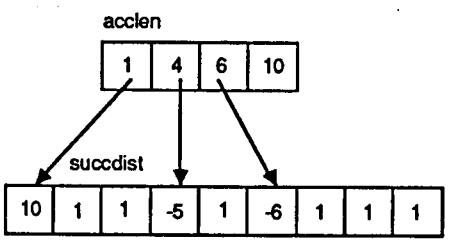

Figure 3: *acclen* and *succdist*.

In other words,

$$succdist[j] = \begin{cases} tail[1] & j = 1 \\ 1 & j \ne acclen[i], \\ & 2 \le j \le M, \\ & 2 \le i \le N-1 \\ tail[i] - tail[i-1] - & j = acclen[i], \\ degree[i-1] + 1 & 2 \le i \le N-1 \end{cases}$$

Now we can compute *succindx*, the indices of all successors, by a linear iteration on *succdist*. In our example, we get the index sequence $10,11,12,7,8,2,3,4,5$.

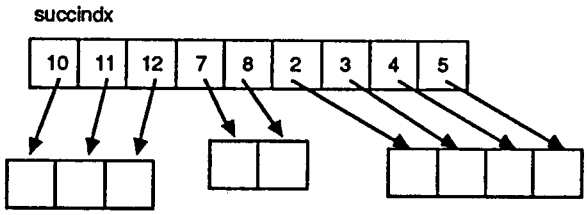

Figure 4: *succdist* and *succ*.

$$succindx[i] = \begin{cases} 1 & i = 1 \\ succdist[i] + succindx[i-1] & 2 \le i \le M \end{cases}$$

Finally, we collect the successors into B, by indexing *succ* with *succindx*.

$$b[i] = succ[succindx[i]],\ 1 \le i \le M$$

Expressed as a program, the algorithm we have described will appear as follows:

```
acclen[1] = 1
for i = 2 to N+1 {
  acclen[i] = acclen[i-1]
    + degree[i-1]
}
M = acclen[N+1] - 1

for i = 2 to M {
  succdist[i] = 1
}

succdist[1] = tail[1]
for i = 1 to N-1 {
  succdist[i] = tail[i] - tail[i-1]
    - degree[i-1] + 1
}
```

```
succindx[1] = tail[1]
for i = 2 to M {
  succindx[i] = succdist[i]
    + succindx[i-1]
  b[i] = succ[succindx[i]]
}
```

As the program shows, the algorithm can be expressed as four completely vectorizable loops, and just four scalar statements.

3 An Example: Parallel Varisized cell Garbage Collection

We will now describe a parallel garbage collection algorithm as an application of the previous algorithm. We state the problem as follows: Given an initial set of node pointers in a vector A, we are to trace down all nodes which can be reached from this initial set. We are then to collect the indices of all cells which *could not* be reached, in a vector *free*.

For the implementation of this algorithm we need an additional vector operation commonly available for supercomputers, namely a *vector compress* operation. Vector compress operations accumulate vector elements which satisfy some condition. The vectorizing compiler for S-820 produces a compress instruction for the following type of code (This code would collect into `b` all elements of `a` which are greater than 17):

```
j = 0
for i = 1 to N {
  if (a[i] > 17) then {
    j = j + 1
    b[j] = a[i]
  }
}
```

We will use a straight forward, mark-and-collect approach:

- Keep a Mark vector, initially zero, with one element for every node.
- Mark all pointers in A.
- Put all the successors of A in a new vector B.
- Compress all unmarked elements of B into A, and repeat from second step until A becomes empty.
- Compress pointers to all unmarked nodes into *free*.

Expressed as a program:

```
; Initialize mark
for i = 1 to MAXNODE {
  mark[i] = 0
}

while N > 0 {

  ; Mark available pointers
  for i = 1 to N {
    mark[A[i]] = 1
  }

  ... <put A's successors in B as
       described above> ...

  ; Compress unmarked
  ; elements into A
  N = 0
  for i = 1 to M {
    if (mark[B[i]] = 0) {
      N = N + 1
      A[N] = B[i]
    }
  }
}

; Collect remaining
; unmarked elements
maxfree = 0
for i = 1 to MAXNODE {
  if (mark[i] = 0) {
    maxfree = maxfree + 1
    free[maxfree] = i
  }
}
```

4 Conclusions

We have described an algorithm for traversing general pointer structures by supercomputer. We have demonstrated how the algorithm easily can be applied to, for instance, parallel varisized-cell garbage collection.

For the implementation of the algorithm, we needed only two special features of the computer: List vector operations, and linear iteration. Both of these operations are available on modern supercomputers. It should be noted here that linear iteration, as it is implemented on pipelined supercomputers, is still a linear operation in terms of computational time complexity. The Connection machine, on the other hand, is able to do linear iteration in logarithmic time, by a parallel prefix operation.

5 Acknowledgments

This work was supported by the Japanese Ministry of Education, and the Swedish National Board for Technical Development. We have benefited very much from discussions with members of the Special Interest Group of the Inference Engine at the university, and with members of the Parallel Programming Systems Working Group at ICOT.

References

[1] Bawden, A., Agre, P.E.: *What a parallel programming language has to let you say.* MIT AI Memo 796. September 1984.

[2] Brooks, R. and Lum, L.: *Yes, An SIMD Machine Can Be Used For AI.* In Proc. of the Int. Joint Conf. on Artificial Intelligence. Los Angeles, 1985. p 73-79.

[3] Hillis, W.D.: *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[4] Hillis, W.D. and Steele, G.L., Jr.: *Data Parallel Algorithms.* CACM, Vol. 29, No. 12. p 1170-1183.

[5] *HITAC S-810 Processor's Handbook.* Manual no. 6010-2-001. Hitachi, Ltd. September 1984. (In Japanese)

[6] Hwang, K. and Briggs, F.A.: *Computer Architecture and Parallel Processing.* McGraw-Hill Computer Science Series. 1986.

[7] Kacsuk, P. and Bale, A.: *DAP Prolog: A Set-oriented Approach to Prolog.* Computer Journal, Vol. 30, No. 5. 1987. p 393-403.

[8] Kanada, Y.: *High-speed Execution of Prolog on Supercomputers.* In Proc. 26th Programming Symp., Information Processing Society of Japan. 1985. p 47-55. (In Japanese)

[9] Kanada, Y.: *High-speed Execution of Prolog on Supercomputers - Realization and Performance of different models of OR-vector execution.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 12, 87-PL-12. July 1987. p 1-10. (In Japanese).

[10] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo. June 1986. p 209-216. Also in Wada, E.(Ed.): Logic Programming '86, Springer LNCS 264. p 170-179.

[11] Nilsson, M. and Tanaka, H.: *A Proposal for implementing GHC on the Connection Machine.* In Proc. IEEE Region 10 Conf. p 821-825. Seoul, August, 1987.

[12] Nilsson, M. and Tanaka, H.: *SIMD Architecture and Superparallel Logic programming.* In Information Processing Soc. of Japan Workshop on Computer Systems, 88-ARC-71. July 1988. Section 71-16.

[13] Nilsson, M. and Tanaka, H.: *A Flat GHC Implementation for Supercomputers.* In Kowalski, R.A, and Bowen, K.A. (Eds.): Proc. Int. Conf. Symp. Logic Programming LP'88. Seattle, August 1988. p 1337-1350.

[14] Steele, G.L., Jr. and Hillis, W.D.: *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing* In Proc. 1986 ACM Conf. Lisp and Functional Programming, Cambridge, Massachusetts. August 1986. p 279-297.

[15] Stolfo, S.J.: *On the Limitations of Massively Parallel (SIMD) Architectures for Logic Programming.* In Proc. US-Japan AI Symp. 1987. ICOT, Tokyo, Japan. December 1987. J. Logic Programming, No. 1, 1986. p 75-92.

[16] Tatsuguchi,K. and Muraoka,Y.: *Parallel Logic Programming Interpreters on Supercomputers.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 14, December 1987. (In Japanese).