

Efficiency of Data Alignment on Maspar*

Bill Maniatty and Boleslaw Szymanski
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180-3590

Balaram Sinharoy
Enterprise Systems Division
IBM Corporation, P.O. Box 950
Poughkeepsie, NY 12602

1 Introduction

Optimal placement of data and operations in large scientific computations executed on a distributed memory machine can significantly reduce the total computation time. In this paper we consider the impact of data and operation alignment on efficiency of computations that arise in theoretical population biology. The computation reported here models an isolated biological system involving interactions between four species: two types, α and β , of plants, their parasite and a virus. Plants populate a field that is subject to winds prevailing in one direction. Parasites prefer to feed on α hosts and will attack a β host only if α hosts are not available. Viruses attack plants but do not affect parasites which carry the virus.

There is significant interest in determining the steady state population densities of different species in the habitat. To simulate the dynamics of such multi-species system, the field is usually partitioned into a grid of small sites, with each site capable of hosting at most one individual of the hosts (plants in our model). Each site is then assigned a state defined by the presence/absence of its potential habitants. A probabilistic model¹ can be used to describe a site's state transition as a function, f , of the states of the neighboring sites. For simplicity, the neighborhood is defined as a rectangular collection of sites with the affected site located inside it (but not necessarily at the center). The size of this rectangle and the placement of the affected site within the rectangle is defined by the ability of the species to spread themselves (or their seeds, in case of plants, or their carriers, in case of viruses). Hence, these parameters depend on the physical characteristics of the species involved and the strength of the wind.

Let n_1, n_2 stand for the lengths of the sides of the affecting rectangle of sites, p_1, p_2 be the offsets (measured from the lower left corner of the rectangle) of the affected site, and $s_{i,j}$ be the current state of the site (i, j) . Then, the state transition probabilities of site (m, q) are defined as:

$$\sum_{i=m-p_1}^{m+n_1-p_1} \sum_{j=q-p_2}^{q+n_2-p_2} f(s_{m,q}, s_{i,j})$$

The above summation is evaluated for each site in the field and for each step of the simulation. Hence, it is likely to dominate the total execution time of the modeling. This is just one example of a reduction operation evaluated simultaneously for consecutive array elements over continuous

*This work was partially supported by National Science Foundation through grants CCR-8920694 and CDA-8805910 and by a grant from the IBM Corporation.

¹The authors are thankful to Prof. Tom Caraco of SUNY at Albany for providing the mathematical model for state transitions.

sections of the same array. Other examples of usage of such operations are likely to be found in modeling physical phenomena (e.g. solvers of partial differential equations characterizing fluid flow).

In this paper we consider implementation of such reductions on a distributed-memory parallel computer where the efficiency of the implementation is strongly affected by the way data structures and operations in the program are distributed over the processors. Operations involving many operands can be performed most efficiently if the executing processor is selected as “close” as possible to the processors that store these operands (closeness is defined by the computer architecture and topology of the network involved). We refer to the problem of distributing data structures and operations as *data alignment*. This problem is particularly acute when the communication is synchronous as in the case of SIMD machines.

Optimization of data alignment for regular iterative algorithms has been analyzed in [1], which also provides an approximate solution for this case. In this paper, we report on data alignment for reduction operations evaluated for consecutive array elements over continuous sections of the same array. As usual, we assume that the binary operator used in reduction is associative and commutative. Some algorithms proposed here also require the existence of an inverse operation, a condition satisfied by the most important reductions involving addition or multiplication. We also assume that an m -dimensional grid, where m is the dimensionality of the array involved in the reduction operation, can be nested in the computer communication network. Under this assumption, we need to consider only reduction operations evaluated simultaneously for an interval (i.e. a one-dimensional, continuous section of a linear array) over the intervals of the same array. The more general, m -dimensional interval case can be solved by applying a one-dimensional interval algorithm to each dimension separately. The presented algorithms are parameterized by the size of the reduced interval n and the position of the final result p . We assume that these parameters change from application to application, but are known at compile-time. Efficiency of the presented algorithms has been measured on Maspar and the results are presented in Section 3.

2 Simultaneous Parallel Reduction

In this section we sketch algorithms designed for simultaneous reduction operations over a one-dimensional, consecutive section of an array, called here an *array interval*. As explained above, we call this operation simultaneous reduction, because each array element is used as an operand to many reductions evaluated over different intervals.² The efficiency of this operation is a function of:

operation cost, i.e., the number of the required reduction operation steps (operation count),

communication cost, i.e., the number of messages sent (message count), the distances traveled by messages (hop count) and the length of the messages (message size), and

memory cost, i.e., the number of memory location used to store intermediate results (memory count).

The communication cost depends on many architectural details of the considered computer. In a typical distributed memory machine, including the Maspar SIMD computer used for implementation

²This is distinct from what is usually referred to as *Parallel Reduction*, which involves parallel evaluation of a single reduction.

and timing of the presented algorithms, the cost of a single communication step can be expressed as

$$start_up_cost + (number_of_hops) * (message_size) * (single_hop_cost)$$

If there are s messages of unit length sent during reduction operation execution, and if i -th message makes h_i hops to reach the destination, then the total communication cost is

$$start_up_cost * s + (single_hop_cost) * \sum_i h_i$$

In many computers the message communication cost is independent of the number of hops made by the message, and in such cases hop count has no impact on the algorithm's efficiency. Nevertheless, to make our analysis more general we consider for each algorithm: operation, memory, message and hop counts as well as message size. For a given interval of size n , the lower limits[2] for these counts are:

$\lceil \log_2 n \rceil$ - for the operation, message and memory counts,

$n - 1$ - for the hop count,

1 - for the message's size.

We consider only algorithms for which each of these counts is no bigger than the double of the corresponding limit.

For an arbitrary offset p and an interval size $n = 2^k$, the well known *Parallel Prefix* algorithm achieves the above limits. Starting with the *distance* equal to 1, and doubling it after every step, each processor:

1. sends its *local value* to the processor *distance* away,
2. applies the reduction's binary operator to the local and received values and stores the result in its *local value*.

The direction of the message transfer is defined in each step by the corresponding bit of the binary representation of the offset p .

For an arbitrary interval size n and an arbitrary offset p , we have designed algorithm called *Intersect* [2] that achieves the minimum communication and memory counts and is within a factor of 1.5 of the minimum operation count. The algorithm splits the reduced interval into two intersecting subintervals with sizes that are powers of 2. Then, our algorithm uses the parallel prefix algorithm on those subintervals. Finally, it adjusts (without additional communication) the result to account for the subinterval's intersection. Adding the additional transfer of the result to an arbitrary offset p increases the communication cost by increasing the message count by one, and the hop count by at most $\lfloor n/2 \rfloor$.

For an arbitrary interval size n and an arbitrary offset p we have designed algorithm called *Split* [2] which produces the result with the minimum memory and hop counts and minimum message size. The operation and message counts are at most the double of the corresponding minimum. The algorithm adds the values in the subintervals defined by the position of the result, sharing operations and communication in both parts as much as possible. Depending on the relative cost of the increased message and operation counts versus smaller hop count, this algorithm may or may not outperform *intersect* for the given interval and offset.

We have also designed an algorithm that, for a given interval size, generates a reduction algorithm that requires asymptotically small operation and message counts (both asymptotic counts of the generated algorithm are $\log_2 n + (\log_2 n)^c + o((\log_2 n)^c)$, where $c = \log_6 3 \approx 0.613147\dots$) [2].

To avoid any conditional operations based on the interval size and the offset at run-time, we have designed a simple code generation tool. This tool takes the given interval size and the offset and produces MPL³ code that implements the proper algorithm for this interval.

3 Results of Execution on Maspar

We have implemented and measured four different algorithms: *sequential*, *commutative*, *split* and *intersect* for simultaneous parallel addition over two-dimensional intervals. The sequential algorithm does not distribute operations or data. Each element in the two-dimensional interval is sent to the processor that needs the result. The commutative algorithm distributes operations to all processors in the interval, but addition is done sequentially across all rows and the last column. The result is obtained in the corner of the two-dimensional interval. The intersect and split algorithms apply the one-dimensional algorithms described above first to rows and then to columns.

The algorithms were run for intervals of various sizes, for 1000 iterations⁴. Timings were taken using the MasPar DpuTimer, and monitored only the summation portion of the computation. The operations column is in units of an operation which performs 9 serial additions on each processing element (this is the number of different states in our biological model). Hop and message counts record Maspar “xnet-shift” operations.

Algorithm	Rows	Columns	Hops	Messages	Memory	Operations	Time
Sequential	2	15	2,250,000	290,000	2	30,000	77.359
Commutative	2	15	1,060,000	150,000	2	16,000	38.713
Intersect	2	15	150,000	50,000	7	6,000	13.772
Split	2	15	150,000	70,000	7	7,000	17.063
Sequential	5	7	1,750,000	340,000	2	35,000	99.248
Commutative	5	7	310,000	100,000	2	11,000	32.526
Intersect	5	7	100,000	60,000	8	7,000	15.885
Split	5	7	100,000	70,000	7	7,000	16.920
Sequential	8	8	4,480,000	630,000	2	64,000	178.149
Commutative	8	8	560,000	140,000	2	15,000	45.419
Intersect	8	8	140,000	60,000	7	6,000	14.730
Split	8	8	140,000	60,000	6	6,000	15.090
Sequential	25	25	150,000,000	6,240,000	2	625,000	1979.300
Commutative	25	25	6,000,000	480,000	2	49,000	138.536
Intersect	25	25	480,000	100,000	13	12,000	27.004
Split	25	25	480,000	120,000	12	12,000	29.297

References

- [1] B. Sinharoy and B.K. Szymanski, “Data Alignment for SIMD Machines,” submitted to *IEEE Trans. Parallel and Distributed Systems*, also Rensselaer Polytechnic Institute, Tech. Rep. CS 91-10, May 1991.
- [2] B.K. Szymanski et.al., “Simultaneous Parallel Reduction,” Rensselaer Polytechnic Institute, Tech. Rep. CS 92-32, October 1992.

³MPL is a dialect of the C programming language used on the MasPar

⁴The time of Sequential algorithm for the largest interval was estimated from runs for 50 and 100 iterations.