計算機アーキテクチャ 95−13
(1992. 8. 19)

高並列計算機 AP1000 のノーマルハイパーキューブアルゴリズムの性能

I. Chuang, 堀江 健志
(株) 富士通研究所
monkey@mit.edu, lions@flab.fujitsu.co.jp

1992年7月22日

N^2 個のプロセッサを仮定したノーマルハイパーキューブアルゴリズム $\mathcal{A}$ は $N\times N$ 構成の高並列計算機 AP1000 で $2N$ フェーズでシミュレーションできることを示す．ノーマルトーラスアルゴリズムに対して $\mathcal{A}$ はメッセージ数がすくないので，通信オーバヘッドが小さい．このため多くの場合，ノーマルトーラスアルゴリズムに比べ性能が良い．本論文では $\mathcal{A}$ を用いたグローバル演算，スキャン（プリフィックスの計算），および全対全通信の性能について検討する．

Normal Hypercube Algorithm Performance on the AP1000

Isaac Chuang　　Takeshi Horie
Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku
Kawasaki 211, Japan
monkey@mit.edu, lions@flab.fujitsu.co.jp

We show that the simulation of a N^2 processor Normal Hypercube Algorithm $\mathcal{A}$ on a $N\times N$ processor Fujitsu AP1000 torus may be performed in $2N$ cycles. Our results indicate that $\mathcal{A}$ is often faster than the corresponding normal torus algorithm, because the number of messages passed is lower and thus overhead is reduced. We present performance data for global reduction, scan (prefix calculation), and all-to-all personalized communications.

1 Introduction

Normal hypercube algorithms constitute an important class of algorithms used to deal with parallel prefix calculation, global reduction, fast Fourier transforms, and sorting, among other problems common to distributed memory parallel processors. The ability to efficiently simulate normal hypercube algorithms is important in the design and evaluation of potential network structures.

In this paper, we investigate the simulation of normal hypercube algorithms on the Fujitsu AP1000's two-dimensional toroidal mesh interconnect. In Section 2, we define the problem in detail and estimate the expected performance. We also propose an optimal node mapping function based on the "crossover" network. Next, we present experimental results comparing the use of normal hypercube algorithms to the usual torus algorithms for several problems including global reduction, prefix calculation, and all-to-all personalized communications. Finally, we conclude with an evaluation of the usefulness of normal hypercube algorithms on the AP1000.

2 Normal Hypercube Algorithms on a 2D Torus

The problem we consider here is the simulation of a normal hypercube algorithm on a N processor, AP1000 (two-dimensional, wormhole-routed, structured-buffer pool) toroidal mesh[1]. The algorithm is based on a N processor hypercube.

Normal hypercube algorithms $\mathcal{A}$ are defined[2] as those which utilize only one dimension of hypercube edges at any step, and moreover, use only consecutive dimensions in consecutive steps. In this paper, without loss of generality, we limit ourselves to the subset $\mathcal{A}'$ which includes only those normal hypercube algorithms which use each dimension once and only once. This simplifies performance estimation.

2.1 Hypercube networks

Our strategy in embedding a normal hypercube algorithm $\mathcal{A}'$ on a torus is to provide a mapping function between the nodes of the hypercube and those of the torus. To do this, we first consider the labeling of nodes in two hypercubic networks, the butterfly and crossover.

The r-dimensional butterfly network, which is quite similar in structure to the hypercube, has $(r+1)2^r$ nodes labeled by pairs $\langle w, i \rangle$, where i is the *level* of the node ($0 \leq i \leq r$) and w is a r-bit number denoting the node's *row*. Two nodes $\langle w, i \rangle$ and $\langle w', i' \rangle$ are linked if and only if $i' = i + 1$ and either $w = w'$ or $w' = w \otimes 2^{i'}$ ("$\otimes$" denotes an exclusive-or operation).

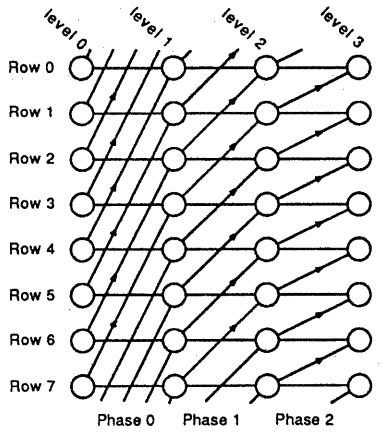

図 1: N=8 crossover network which is computationally equivalent to the butterfly network.

The r-dimensional crossover[1] is a multistage interconnect network (MIN) similar to the butterfly; only the placement of cross edges differs. Two nodes $\langle w, i \rangle$ and $\langle w', i' \rangle$ are linked if and only if $i' = i + 1$ and either $w = w'$ or $(w + 2^i) \bmod 2^r = w'$. A 3-dimensional crossover network is shown in Figure 1. The crossover is not isomorphic to the butterfly, but is of interest for its unidirectional communication pattern (it maps well onto a torus).

[1]The crossover is similar to the network used in [3].

Note that a N-node normal hypercube algorithm $\mathcal{A}'$ may be simulated trivially on a $\log N$ dimensional butterfly network, by mapping the $\log N$ steps of $\mathcal{A}'$ to consecutive levels i of the butterfly. Likewise, $\mathcal{A}'$ also may be simulated trivially on the crossover. Visualization of $\mathcal{A}'$ on a butterfly or hypercube is easy, because $\mathcal{A}'$ is simply a calculation which traverses the MIN once through, for example, from left to right.

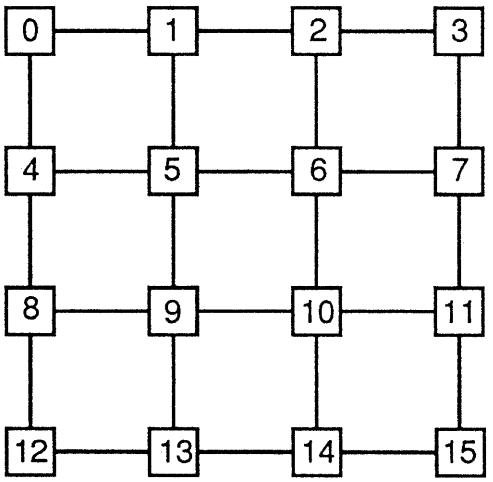

図 2: Raster scan labeling of nodes in a two dimensional torus.

2.2 The AP1000 Torus

The $a \times a$ two-dimensional torus has a^2 nodes, each of which is connected via one physical channel to its four nearest neighbors. The diameter of this network is a. *Wormhole routing*[4] allows m-flit messages to be passed between d-hop nodes in time proportional to $(m/B + d)$, where B is the bandwidth of the physical channel. *Structured-buffer pools*[1] are used to improve throughput[2] by allowing the bandwidth B of a single physical channel to be distributed between contending messages such that d messages sharing the same channel each receive a bandwidth of about B/d.

We also assume that routing on the torus is performed unidirectionally and statically. All messages are first routed in the $+x$ (east) direction, then in the $-y$ (south) direction after arriving at the appropriate column. Routing is handled entirely by the hardware, and does not involve any processor interaction except for message generation and reception. Although the AP1000 may also route messages bidirectionally, it does so with less efficiency[6].

We label the nodes of the torus by drawing the torus as a mesh with wraparound connections, then beginning with the top left node as number 0, we enumerate nodes in raster-fashion through the entire array. This labeling scheme is pictured in Figure 2.

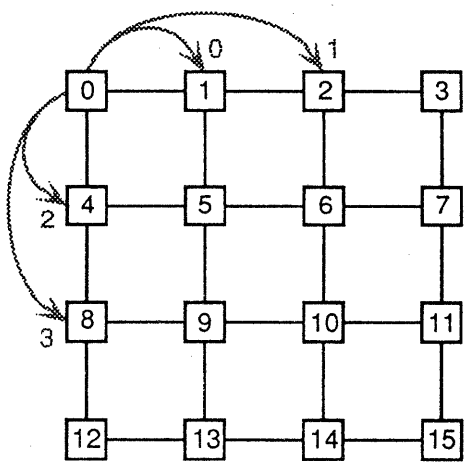

図 3: 4×4 torus showing the four messages passed from node 0 during the four phases in simulating a r=3 crossover network. During the first 2 phases, messages are passed in rows, and during the last 2 phases, in columns. All nodes send a message in each phase; in phase i node j sends to node $j + 2^i$. Wraparound paths forming the torus are not drawn.

[2]This works in a similar manner to *Virtual Channels*[5]

Finally, we define a *normal torus algorithm* $\mathcal{A}_t$ as one which operates on a $a \times a$ torus in $2a$ phases in the following manner. In the first a phases, information is passed from east to west between all neighboring nodes, and in the remaining a phases, information is passed from north to south. As usual, at each step computation may be performed at each node. Normal torus algorithms are often used to perform sorting, global reduction, and other calculations on a two-dimensional torus.

2.3 Mapping Functions

A N-node normal hypercube algorithm $\mathcal{A}'$ may be executed on the AP1000 torus by mapping hypercube node u to torus node $v = f(u)$, where the function $f(x)$ is the identity function $f(x) = x$. Synchronization between phases in $\mathcal{A}'$ is performed implicitly, because all nodes in the torus participate, and all nodes must receive messages at each step in the algorithm.

Alternatively, we may choose a mapping function based on the butterfly or crossover networks, whereby node $\langle w, i \rangle$ is mapped to node w of the torus at phase i in $\mathcal{A}'$. In this case, note that only cross edges require message passing; straight edges are all internal. Thus, the number of messages passed at each step in $\mathcal{A}'$ is N, which is optimal.

2.4 Theoretical Performance Comparison

Given the mapping function defined in the previous sections, we first postulate that *embedding of the crossover communication pattern in the torus will achieve superior performance to that for a butterfly, because of its more uniform traffic*; physical channels between nodes on the torus are shared between fewer messages when scans are generated from the crossover rather than the butterfly MIN[3]. We believe this postulate is obvious by inspection, and provide experimental verification in Section 3.1.

Second, we postulate that *a normal hypercube algorithm $\mathcal{A}'$ running on an AP1000 torus will run at least as fast as the equivalent normal torus algorithm $\mathcal{A}_t$ achieving the same purpose* (global reduction, sorting, etc.). In fact, $\mathcal{A}'$ will usually run significantly faster. To show this, we first estimate the number of cycles needed to perform the communication needed when embedding $\mathcal{A}'$ on a torus.

Without loss of generality, consider the embedding of a $2n$-dimensional crossover network in a rectangular torus with $N = 2^n \times 2^n$ nodes. During phase i $(0 \leq i < 2n)$, messages are passed from node j to node $j + d_i$, where $d_i = 2^i$. Phase i corresponds to the communication between levels i and $i+1$ of the crossover, and torus node j is mapped to the crossover nodes $\langle j, i \rangle$, as explained in the last section. Because of the raster-labeling of the torus nodes, it happens to be that messages sent during the first n phases are routed mainly along rows, while those sent during the second n phases are routed along columns only. This is shown in Figure 3.

Because wormhole routing is used, the number of cycles needed to pass messages between two nodes is largely independent of the distance between the nodes. Thus, message latency depends primarily on the length of the message being sent, and the maximum number of messages sharing a required physical channel. We define $T_0 = L/B$ as the number of cycles required to send a message of length L between nodes over a channel with bandwidth B in the absence of network congestion.

In our case, when node j sends to $j + d_i$ for $0 \leq i < n$, $d_i - i$ other nodes will also be sending messages along the same path. Thus, the effective bandwidth is degraded from B to

[3]Note that this distinction only matters for tori with unidirectional interconnects; with bidirectional routing there should be no difference in the performance of the crossover and butterfly mappings

approximately B/d_i during the first n phases. During the second n phases, $n \leq i < 2n$, $d_i/2^n$ nodes send messages down each column, giving an effective bandwidth of approximately B/d_{i-n}.

Taking into account network congestion, therefore, we find that the total number of cycles τ_t needed to simulate a crossover on the torus is approximately given by:

$$\tau_t = \frac{T}{T_0} = \sum_{i=0}^{n-1} d_i + \sum_{i=n}^{2n-1} d_{i-n}\,, \qquad (1)$$

$$\tau_t = 2(2^n - 1) \approx 2\sqrt{N}\,. \qquad (2)$$

Note that this expression does not hold strictly when L is small, because for small L, distance becomes important in calculating message latency, and also, network congestion degradation factors become harder to estimate. For example, when $L = 1$ flit, no congestion occurs because one flit is exchanged between nodes each cycle and no messages ever share the same physical channel[4]

The significance of Eq. (2) is clear; it shows that a normal hypercube algorithm $\mathcal{A}'$ simulated on a square, N-node AP1000 torus using a crossover generated scan runs in approximately the same time (number of communication *cycles*) as for a normal torus algorithm $\mathcal{A}_t$. Furthermore, only $\log_2 N$ *phases* are required to simulate $\mathcal{A}'$, compared to the $2\sqrt{N}$ phases needed for $\mathcal{A}_t$; this is possible because hardware routing mechanisms are used to distribute limited resources. Reducing the number of required phases reduces the overhead incurred in calling message passing library functions because fewer messages are sent.

3 Benchmark Results

We evaluated the performance of normal hypercube algorithm simulation on the Fujitsu AP1000. In this section, we first present data comparing the performance of the different communication patterns discussed above, then summarize results for global reduction, scans (prefix calculation), and all-to-all personalized communications.

3.1 Communication Kernels

We evaluated the execution time of three different communication patterns on the AP1000, varying the message size and the number of processors. The code kernels are described below.

The **butterfly** kernel exchanges messages between all N processors in $\log_2 N$ phases. In the code below, `n` is the number of processors, `len` is the length of the message exchanged in bytes, and `cid` is the raster-scan ID-number assigned to each processor.

```
for(i=1;i<n;i<<=1){
    SendMsg(cid^i,msg,len);
    RecvMsg(cid^i);
}
```

The butterfly kernel is expected to perform worse than the **crossover** kernel, listed below, which exchanges messages between processors in a unidirectional pattern, thus allowing fewer physical channels to have to be shared between messages. Both the butterfly and crossover may be used to simulate a normal hypercube algorithm on the AP1000. `mask` is set to `n-1`, where `n` is understood to be an integer power of two.

```
for(i=1;i<n;i<<=1){
    SendMsg((cid+i)&mask,msg,len);
    RecvMsg((cid-i)&mask);
}
```

Communication for the normal torus algorithm is performed using the code kernel below. Messages are first passed between all processors on each row, then on each column. `nx` and `ny` are the number of processors in each dimension, and `nx+ny` phases are performed.

[4]Note that Eq. (2) is a direct generalization of the results of Nassimi and Sahni[7].

```
for(i=0;i<nx;i++){
    SendEast(msg,len);
    RecvWest();
}
for(i=0;i<ny;i++){
    SendSouth(msg,len);
    RecvNotrh();
}
```

Execution times for the three kernels are shown in Figure 4. For short messages, the crossover network always performs best, as expected.

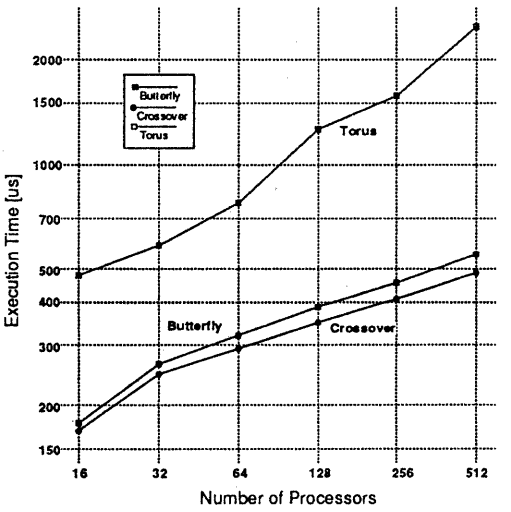

図 4: Execution times of the four communication kernels. The message size is fixed at 16 bytes. User-level message passing library functions were used.

3.2 Global Reduction

Global reduction is the calculation of $y = x_0 \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_i \otimes \cdots \otimes x_{N-1}$, over the values x_i held in the N processors, and the redistribution of the result y to all processors. In general, $\otimes$ may be any binary operation.

Figure 5 compares the performance of a global sum function implemented using two different communication patterns: crossover, and binary

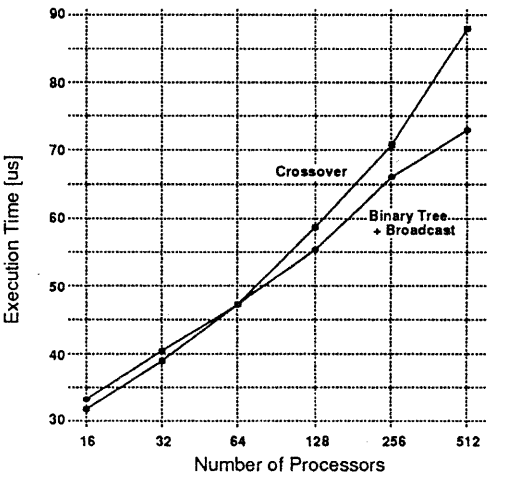

図 5: Performance of the global sum function on the AP1000 as a function of the number of processors. The crossover and binary tree+broadcast algorithms are compared. The algorithms were implemented using system-level calls to native machine I/O.

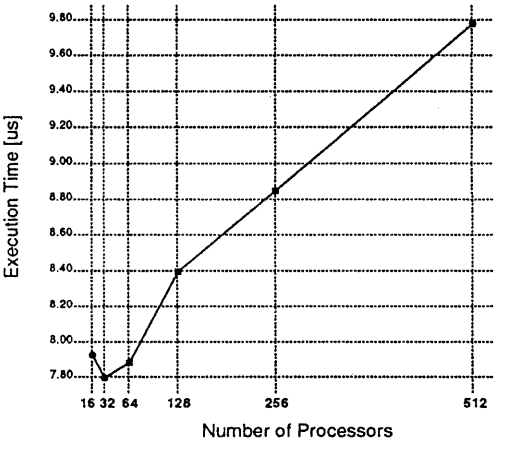

図 6: Execution time per phase $t_{\text{phase}} = t_{\text{total}}/\log_2 N$ for global summation using the crossover network.

tree. The binary-tree algorithm uses $\log_2 N$ phases like the crossover, but requires an additional step at the end to broadcast the final result to all the processors. The crossover algorithm performs better than the tree until the number of processors exceeds 64, because the crossover suffers more contention than the tree does[5].

The large difference in execution times shown in Figure 4 and those in Figure 5 arises as a result of using user-level as opposed to system-level message passing functions[6]. Figure 6 shows the time *per phase* calculated from data for the crossover network from Figure 5. The linear increase for large N is due to increased contention in the network.

3.3 Scan (Parallel Prefix)

The scan operation[9], otherwise known as parallel prefix calculation[3], is widely used in hypercube algorithms, and in numerical codes for solving tridiagonal equations, among other problems. Mathematically, prefix calculation is defined as the calculation of all y_i, for $0 \leq i < M$, and $y_i = x_0 \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_{i-1} \otimes x_i$. In particular, this calculation arises in loop 5 of the Livermore Fortran Kernels[10] benchmark suite. We find that using the crossover network, the AP1000 can achieve a peak of 12.6 Mflops for this loop (M=1000), using 64 processors. Figure 7 indicates that performance rapidly levels out as the number of operands increases, because operation time becomes dominated by single-processor speed and not interprocessor communication.

[5] Although the message size is fixed (one `double` (8 bytes)), including 8 bytes of header information, this is 4 flits in the current AP1000 implementation

[6] Times in Figure 5 arguably represent the best performance achievable with the AP1000 for the problem considered. It is instructive to compare 64-processor AP1000 and EM-4[8] times for the same communication pattern: global sum 47.3μs vs. 14.4μs (note that the EM-4 time is for $\sum$ `ints`). Reception in the AP1000 is slowed by software message matching.

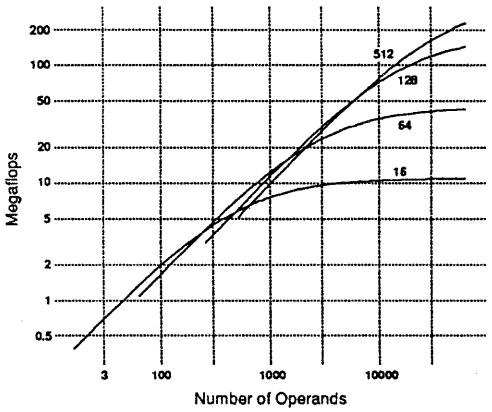

図 7: Performance of the scan operation. Speed of the calculation of partial products of doubles is plotted as a function of the number of operands for 16 to 512 processor configurations. A peak of around 230 Mflops can be attained using 512 processors.

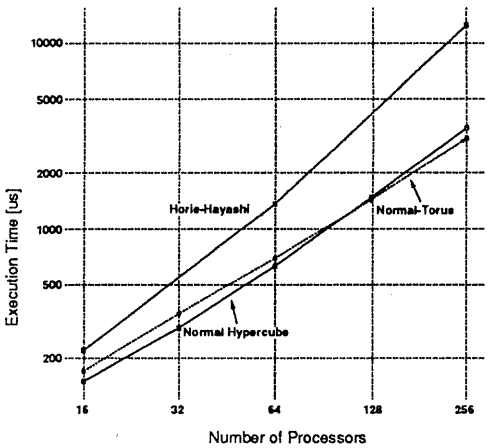

図 8: Comparison of the performance of the Horie-Hayashi ($\mathcal{A}_{\rm hh}$), Normal Hypercube ($\mathcal{A}_{\rm bp}$), and Normal Torus ($\mathcal{A}_{\rm nt}$) algorithms for one-double (8 byte) all-to-all personalized communications on the AP1000.

3.4 All-to-all personalized communications

All-to-all personalized communications is the problem in which every processor desires to send every other processor a unique message. On a two-dimensional torus, this may be performed using the Horie-Hayashi algorithm[6], $\mathcal{A}_{hh}$. Alternatively a bucket-passing normal hypercube algorithm, $\mathcal{A}_{bp}$, may be simulated. The advantage of the latter course is that the number of steps is reduced dramatically, from approximately $N^{1.5}/4$ to $\log_2 N$ for N processors. However, using $\mathcal{A}_{bp}$ requires that $N/2$ messages be exchanged between processors at each step; thus, for large messages, $\mathcal{A}_{bp}$ is inferior to $\mathcal{A}_{hh}$. As Figure 8 shows, the $\mathcal{A}_{bp}$ algorithm is useful for small messages. $\mathcal{A}_{bp}$ and $\mathcal{A}_{nt}$ are always faster than $\mathcal{A}_{hh}$ because the message size is small, and $\mathcal{A}_{bp}$ is faster than $\mathcal{A}_{nt}$ for $N < 128$.

4 Conclusion

We have shown that although it may often seem non-intuitive to use a hypercube-based algorithm on a torus-based architecture computer such as the AP1000, better performance can often be attained using such algorithms rather than regular torus algorithms because the number of communication phases is reduced. We have also presented an optimal mapping function based on the crossover network, which adapts well to unidirecitonally routed tori.

We acknowledge T. Shimizu for his assistance, and thank M. Ishii, H, Shiraishii, H. Sato, K. Hayashi, and the MIT-Japan program for making the collaborative research visit possible.

参考文献

[1] T. Horie, H. Ishihata, and M. Ikesaka. Design and Implementation of an Interconneciton Network for the AP1000. *To Appear (IFIP '92)*, 1992.

[2] F. Thomas Leighton. *Introduction to Parallel Algorithms: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, California, 1992.

[3] W. Daniel Hillis and Jr. Guy L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170, 1986.

[4] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547, May 1987.

[5] W. J. Dally. Virtual-Channel Flow Control. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, page 60. IEEE Computer Society Press, May 1990.

[6] T. Horie and K. Hayashi. トーラスネットワークにおける最適全対全通信方式 (An Optimal All-to-all Personalized Communication Algorithm for Torus Networks). In *JSPP '92*, page 187, June 1992.

[7] Howard J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* McGraw-Hill Publishing Company, New York, NY, 1990.

[8] A. Shaw, Y. Kodama, M. Sato, S. Sakai, and Y. Yamaguchi. Data-Parallel Programming on the EM-4 Dataflow Parallel Supercomputer. In *JSPP '92*, page 179, June 1992.

[9] Guy E. Blelloch. Scan Primitives and Parallel Vector Models. Technical Report MIT-LCS-TR-463, MIT-LCS, October 1989.

[10] Frank McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. *Lawrence Livermore National Laboratory document UCRL-53745*, December 1986.