Architecture Scalability of Parallel Vector Computers with a Shared Memory

Eskil Dekker, *Member*, *IEEE*

Abstract—Based on a model of a parallel vector computer with a shared memory, its scalability properties are derived. The processor-memory interconnection network is assumed to be composed of crossbar switches of size $b \times b$. This paper analyzes sustainable peak performance under optimal conditions, i.e., no memory bank conflicts, sufficient processor-memory bank pathways, and no interconnection network conflicts. It will be shown that, with fully vectorizable algorithms and no communication overhead, the sustainable peak performance does not scale up linearly with the number of processors p. If the interconnection network is unbuffered, the number of memory banks must increase at least with $O(p \log_b p)$ to sustain peak performance. If the network is buffered, this bottleneck can be alleviated; however, the half performance vector length still increases with $O(\log_b p)$. The paper confirms the validity of the model by examining the performance behavior of the LINPACK benchmark.

Index Terms—Architecture scalability, parallel vector computers, shared memory, sustainable peak performance, theoretical peak performance.

1 Introduction

The classical supercomputer concept as it has been employed by several manufacturers can be characterized as a parallel vector computer with a shared memory. Besides using multiple dedicated functional units within a single processor, this approach towards high-performance computing also makes extensive use of pipelining. With vector processing, identical operations on multiple operands can be performed with a single instruction. These techniques provide a significant increase of throughput for a single processor. To permit parallel processing, multiple processors operate concurrently.

The memory bandwidth of the shared memory must be sufficient to supply each processor with operands required to sustain continuous execution. The memory is divided into memory banks which can be accessed in parallel, and the memory is interleaved to further increase the throughput. The memory banks are connected to the processors with an interconnection network. Although this kind of architecture is still employed in current parallel computers, it is widely accepted that it cannot sustain a large number of processors due to scalability problems.

In Bell [3], practical issues of size-, generation-, and problem-scalability are discussed. A formal definition due to [22] focuses on algorithm-architecture scalability. The ratio of speedups of the algorithm on a real machine and a specific theoretical parallel machine (PRAM) defines scalability. An evaluation of this and other algorithm-architecture scalability measures can be found in [16]. Unfortunately, due to the influence of the algorithm characteristics, these measures cannot be used to give a definite statement regarding a specific architecture. Hill [13] argues that scalability should be defined for architectures alone, if possible. Based on limiting technology, [2] investigates the theoretical aspects of architecture scalability without addressing actual realization. Here, architecture scalability is analyzed by considering the hardware implementation. Furthermore, in order to quantify architecture scalability, a comparison of the architecture is not done with respect to some theoretical parallel machine, but the actual behavior of a real machine is compared to its ideal behavior. In order to eliminate the influence of the algorithm characteristics upon the scalability measure, one might consider algorithms that perfectly match the architecture. Obviously, such algorithms should be meaningful; however, performance degrading factors such as a limited degree of parallelism should not arise. The theoretical peak performance is one of the major characteristics of a real parallel machine, so it is natural to define scalability with respect to sustainable peak performance of representative algorithms. An important benchmark for evaluating the performance of computers is LINPACK [8]. This benchmark consists of the solution of a large dense linear system with LU decomposition. Although such a benchmark can never represent a real workload and sustained performance for real codes is only a fraction of the theoretical peak performance [12], LINPACK performance plays a significant role in many numerically intensive applications. Apart from division and square root operations, basic kernels of numerical algorithms consist of square norm, inner product, and saxpy operations. These kernels are fully vectorizable—i.e., peak performance can be obtained in ideal conditions—and, therefore, they are optimal with respect to the architecture of parallel vector computers. Furthermore, since they provide the major performance component in benchmarks, it is important to consider their performance characteristics on shared memory architectures. In this paper the scalability of these kernels is investigated with respect to sustainable peak performance.

• *E. Dekker is with the Faculty of Information Technology and Systems, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands. E-mail: e.dekker@dimes.tudelft.nl.*

Manuscript received 23 Jan. 1996; revised 8 Jan. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106499.

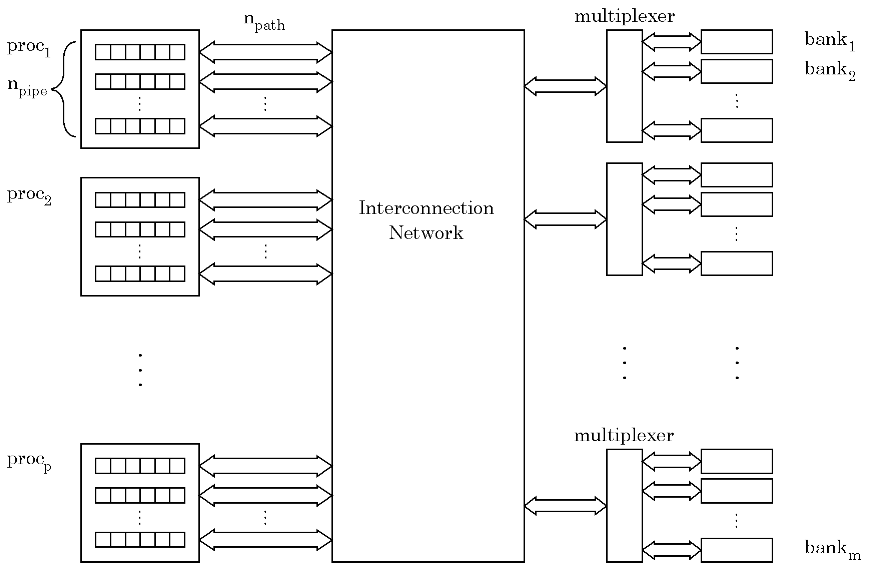

Fig. 1. A parallel vector computer with p processors (proc$_j$) connected with an interconnection network and multiplexers to m memory banks (bank$_j$). Each processor comprises n_{pipe} floating-point units with n_{path} memory pathways.

The sustainable peak performance is analyzed with the reference to the floating-point operations ratio. No other sources of performance degradation, such as interconnection network conflicts or memory bank conflicts, are assumed to exist. It will be shown that the sustainable peak performance is restricted by the maximal memory bandwidth. Moreover, as the number of processors increases, the sustainable peak performance does not scale up proportionally because of constraints imposed by the shared memory. Although other machine characteristics, such as I/O performance, have significant impact upon architecture scalability, they are left out of this discussion.

First, the significant machine parameters of a parallel vector computer are defined. In the next section, its throughput is determined. After that, the actual performance behavior of the considered kernels is derived. Thereafter, the paper treats the actual implementations of shared memory. In order to illustrate the validity of the model, Section 6 analyzes the performance behavior of the LINPACK benchmark.

2 Preliminaries

Fig. 1 depicts a schematic diagram of a general parallel vector computer. The computer has the following features: The machine comprises p processors with a clock cycle τ_{proc}. Each processor consists of n_{pipe} pipelined functional units, which can operate at a rate of one operation per clock cycle τ_{proc} each. Here, only floating-point units are considered since floating-point performance is paramount in scientific computing. There is an equal number of floating-point add and multiply units. For intermediate storage, each processor contains a set of scalar and vector registers. A vector register can contain up to v_l elements.

Each processor has n_{path} pathways to the shared memory, which can transfer operands at a rate of one operand per processor clock cycle each. For each pathway, the processor possesses a memory port which takes care of the addressing and operand transfer between the processor and the shared memory. An interconnection network and multiplexers connect the pn_{path} processor pathways to the m memory banks. The memory-access time, which includes the delay introduced by the interconnection network, the multiplexers, and the memory banks, is equal to τ_{mem}.

Since there are an equal number of add and multiply units, the algorithms must perform exactly one addition per multiplication to reach peak performance. The sample kernels satisfy this requirement. All functional units can be chained so that the smallest vector start-up time can be realized and the least amount of intermediate storage is required.

The kernels require operands from the shared memory. The required operand bandwidth is a function of the reference to floating-point operations ratio R. For large vectors, the values of R are given for the sample kernels in Table 1.

All vectors are stored in the shared memory, the coefficients α and $\alpha_1, \alpha_2, \ldots, \alpha_k$ for the saxpy and the multiple saxpy kernel are initially stored in the registers of a processor. Large vectors cannot be stored in the vector registers within a processor. Therefore, a vector operation has to be split into parts so that corresponding operand vectors fit in

TABLE 1
The Kernels and R,
the Reference to Floating-Point Operations Ratio

Kernel	R	Register storage
$\lVert\underline{x}\rVert^2$	$\frac{1}{2}$	–
$(\underline{x}, \underline{y})$	1	–
$\underline{y}' = \underline{y} + \alpha\underline{x}$	$\frac{3}{2}$	α
$\underline{y}' = \underline{y} + \sum_{j=1}^{k} \alpha_j \underline{x}_j$	$\frac{k+2}{2k}$	$\alpha_1, \alpha_2, \ldots, \alpha_k$

the vector registers. Section 4 addresses this issue in more detail. The kernel code consists of a loop of several vector instructions that is executed repeatedly until the vector operation is completed. It is assumed that the instructions within the loop are stored in an instruction buffer. Since loop control can be performed concurrently with the execution of the loop, no loop overhead exists. Each processor possesses an instruction buffer that is not shown in Fig. 1. The instruction stream is assumed to be solely maintained by the instruction buffer, i.e., no shared memory references are necessary. This assumption is realistic if the instruction storage required for each loop does not exceed the instruction buffer size.

Table 5 shows a concise description of all symbols used in this paper.

3 Throughput

This section shows the dependence of sustainable peak performance within the kernels on the machine parameters. If there are neither interconnection network conflicts nor memory bank conflicts, the maximal available memory bandwidth B is equal to

$$B = \frac{m}{\tau_{\text{mem}}}. \tag{1}$$

Each processor can produce n_{pipe} results per clock cycle τ_{proc}, thus the maximum performance of p processors is

$$P(p) = p\frac{n_{\text{pipe}}}{\tau_{\text{proc}}}. \tag{2}$$

The required memory bandwidth to sustain this performance depends on the reference to floating-point operations ratio R. Table 1 shows that this parameter is kernel dependent. However, the architecture imposes a limit on this parameter. If a processor possesses n_{path} pathways, each capable of transferring one operand per processor clock, R must satisfy

$$R \le \frac{n_{\text{path}}}{n_{\text{pipe}}}, \tag{3}$$

to allow computation bound processing, i.e., the performance is limited by the maximum operation rate of the processor pipes. If

$$R > \frac{n_{\text{path}}}{n_{\text{pipe}}}, \tag{4}$$

then there is transfer bound processing. The throughput is impeded by the limited number of pathways, and the peak performance can never be reached. In the sequel, it is assumed that there are enough pathways to supply a processor running at peak rate. It is assumed that there are enough vector registers per processor to implement a kernel, and that each processor executes the same kernel with different data so that neither communication nor synchronization is required between the processors. The required memory bandwidth B_{req} for p processors is equal to

$$B_{\text{req}} = RP(p). \tag{5}$$

The required memory bandwidth cannot exceed the available memory bandwidth.

$$B_{\text{req}} \le B, \tag{6}$$

or

$$pR\frac{n_{\text{pipe}}}{\tau_{\text{proc}}} \le \frac{m}{\tau_{\text{mem}}}. \tag{7}$$

The maximum number of processors running at peak performance is thus bounded by

$$p \le \frac{1}{R}\frac{\tau_{\text{proc}}}{n_{\text{pipe}}}\frac{m}{\tau_{\text{mem}}}. \tag{8}$$

With (2) and (8), the maximum achievable performance $P_{\max}$ is

$$P_{\max} = \frac{1}{R}\frac{m}{\tau_{\text{mem}}}. \tag{9}$$

Not surprisingly, the maximum achievable performance is proportional to the memory bandwidth. From this it is clear that kernels such as polynomial evaluation with a small value of R, and $R \le \frac{n_{\text{path}}}{n_{\text{pipe}}}$, can achieve a higher sustainable peak performance than the sample kernels provided that there is no limit upon the number of processors.

4 Performance Behavior

For kernels with large vectors, the vector start-up time can be neglected. In order to determine the complete performance behavior, the vector start-up time must be considered as well. Therefore, the timings of the kernels are derived under ideal conditions.

Other assumptions are required for scheduling and vector register allocation. It is assumed that a single functional unit supports multiple vector operations. As soon as the first stage of a pipelined functional unit has processed the last elements of a vector operation, the next vector operation can be issued. Although the next vector operation involves a different set of registers, no additional delay caused by pipeline reconfiguration is assumed. Source and destination registers of a single vector operation must differ in order to prevent source-destination conflicts. However, consecutive vector operations on the same functional unit can use a destination register of the first operation as source register for the second operation. In this way, the common register of consecutive vector operations acts as a delay line.

First, the saxpy kernel is investigated. The vectors $\underline{y}'$, $\underline{y}$, and $\underline{x}$ with vector length v_{kernel} are split into the vector parts $\underline{y}'^{(i)}$, $\underline{y}^{(i)}$, and $\underline{x}^{(i)}$, for $i = 1, 2, \ldots, \left\lceil \frac{v_{\text{kernel}}}{v_l} \right\rceil$, with a maximal vector length v_l. Fig. 2 shows the timing of two consecutive updates $\underline{y}'^{(i)}$ and $\underline{y}'^{(i+1)}$ on one processor with one add and one multiply unit.

In Figs. 2, 3, and 4, the timing of each vector operation is indicated by a bar. The consecutive squares inside the bar represent the vector elements of the destination register. For the vector store operation these squares represent the different memory banks. Consecutive results appear in consecutive

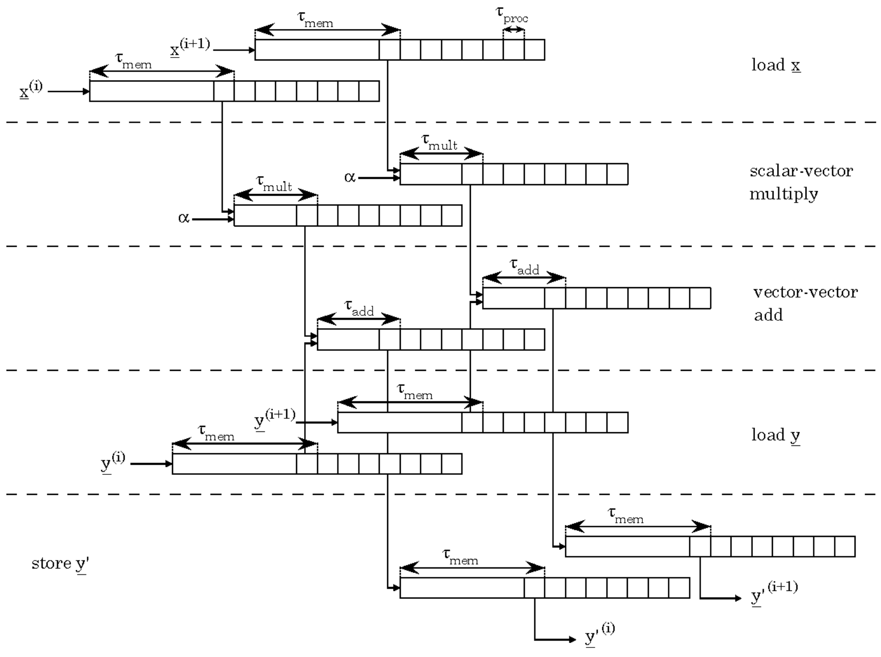

Fig. 2. The saxpy kernel implemented with one add and one multiply unit. Two updates for the parts $\underline{y}'^{(i)}$ and $\underline{y}'^{(i+1)}$ are shown.

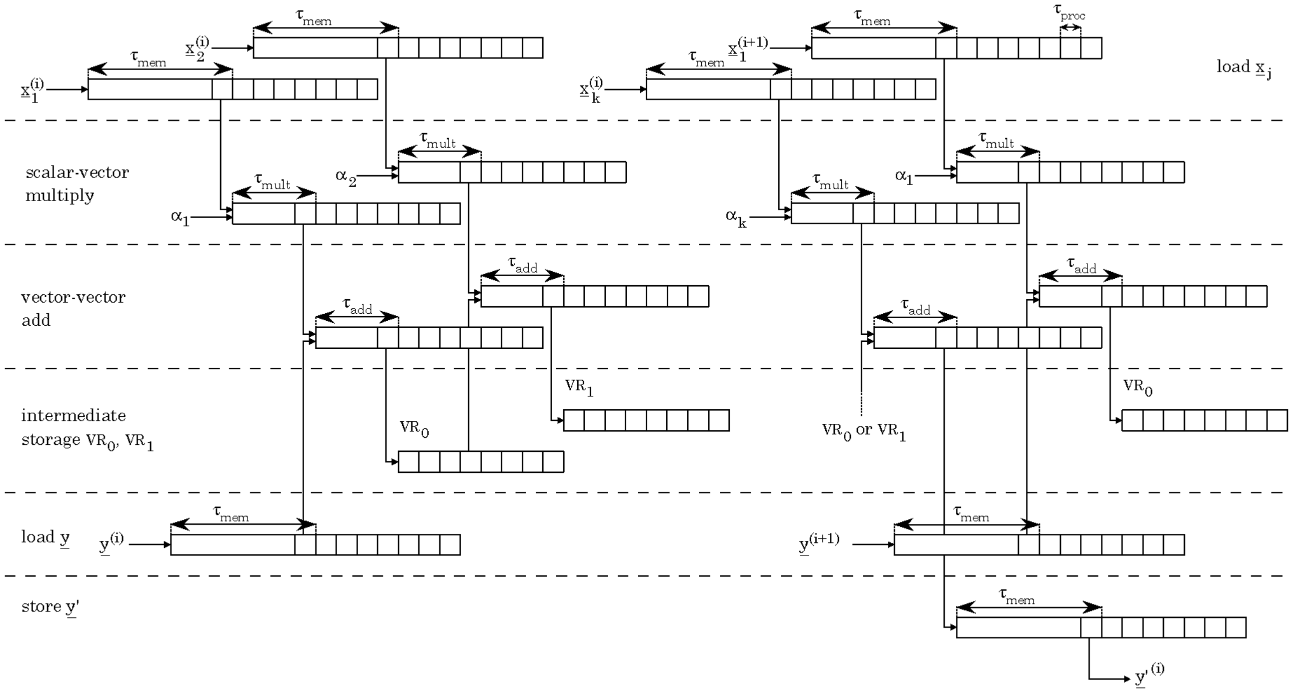

Fig. 3. The multiple saxpy kernel implemented with one add and one multiply unit. The complete update of part $\underline{y}'^{(i)}$ and the first update of part $\underline{y}'^{(i+1)}$ are shown.

squares to the right. There is an initial delay for the first result to be available that depends on the kind of operation. That is, for a vector-vector addition, the initial delay is τ_{add}. After that, a next result is available after each τ_{proc}.

For the update $\underline{y}'^{(i)} = \underline{y}^{(i)} + \alpha\underline{x}^{(i)}$, first the vector part $\underline{x}^{(i)}$ is loaded from shared memory, indicated by the lower left bar in the load $\underline{x}$ segment of Fig. 2. After a delay τ_{mem}, the first vector element of $\underline{x}^{(i)}$ is available, indicated by the first square from the left inside this bar. No memory reference is needed for α because it was initially stored in a register. By chaining the vector load with the scalar-vector multiplication $\alpha\underline{x}^{(i)}$, the first result of this multiplication is available after

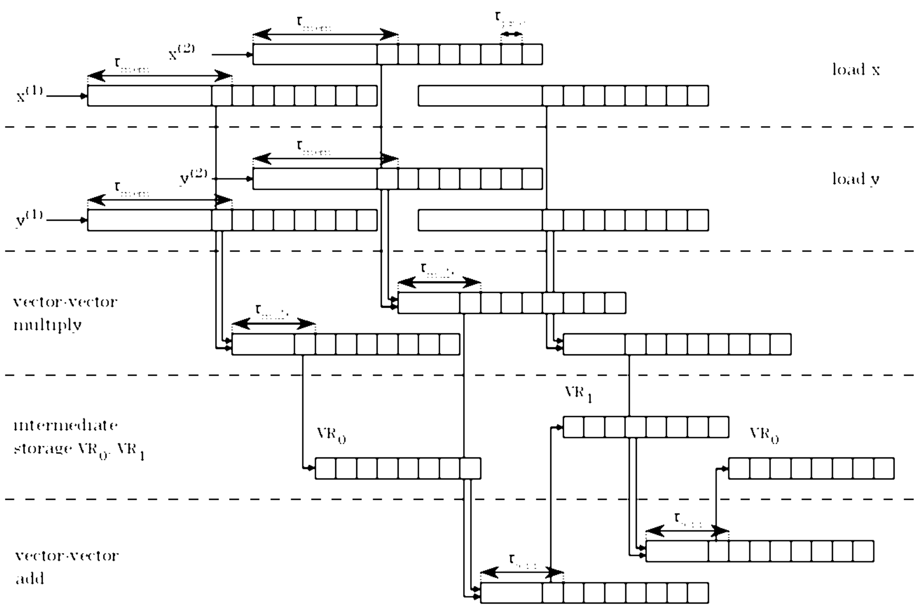

Fig. 4. The inner product kernel implemented with one add and one multiply unit. The calculation of two partial sum vectors $(\underline{x}, \underline{y})^{(1)}$ and $(\underline{x}, \underline{y})^{(2)}$ are shown.

an additional delay τ_{mult}. The lower left bar of the scalar-vector multiply segment represents this multiplication. In order to prevent any delay in the following vector-vector addition $\underline{y}^{(i)} + \alpha\underline{x}^{(i)}$, the vector part $\underline{y}^{(i)}$ is loaded from shared memory so that the first element of $\underline{y}^{(i)}$ is available at the same time the first multiplication result is available. In Fig. 2, the first squares from the left in the lower left bars of the load $\underline{y}$ and the scalar-vector multiply segments exactly coincide. After a delay τ_{add} the first result of the chained vector-vector addition is available and it can be written back to shared memory, indicated by the lower left bar in the store $\underline{y}'$ segment. The vector operations for the next vector update $\underline{y}'^{(i+1)} = \underline{y}^{(i+1)} + \alpha\underline{x}^{(i+1)}$, which are represented in Fig. 2 by the upper right bars in each segment, are scheduled so that each first result is available just after the last result of the corresponding vector operation of the $\underline{y}'^{(i)} = \underline{y}^{(i)} + \alpha\underline{x}^{(i)}$ update.

Intermediate storage of the vector $\underline{y}'$ is used to exploit the low value of R for the multiple saxpy. Each vector part $\underline{y}^{(i)}$ is updated with all $\underline{x}_j^{(i)}$ for $j = 1, 2, \ldots, k$, before the next part $\underline{y}'^{(i+1)}$ is computed. The timing of the multiple saxpy is shown in Fig. 3.

Two vector registers, VR_0 and VR_1, serve as intermediate storage for $\underline{y}'^{(i)}$. If k is odd, the last vector addition involves VR_1 as source register; otherwise, it uses register VR_0. The update of $\underline{y}^{(i+1)}$ begins as soon as the first pipeline stages of the functional units are available. No additional vector start-up time is needed since the vector operations can be overlapped.

The inner product of two vectors $\underline{x}$ and $\underline{y}$ with vector length v_{kernel} can be done in a similar manner. In this case, the vector of products of $\underline{x}^{(i)}$ and $\underline{y}^{(i)}$ is computed. While the next pair is being multiplied, the previous product vector is added to an intermediate sum vector. At the completion of the operation, vector register VR_0 contains a sum vector of v_l elements if $\left\lceil \frac{v_{\text{kernel}}}{v_l} \right\rceil$ is odd, otherwise, the sum vector resides in VR_1. The inner product, which is equal to the sum of elements, is determined with $v_l - 1$ scalar additions. Afterward, the result is written back to the shared memory. These operations are not shown in Fig. 4. For the square norm kernel the timing is identical with the exception that the load $\underline{y}$ segment in Fig. 4 is superfluous.

If there are more floating-point add and multiply units per processor, the actual implementation of the kernels does not change. In this case, consecutive operations within a vector operation are allocated to different units. For example, if there are two add units, a vector addition is split into two: The even elements of the vectors are processed by one unit and the odd elements are processed by the second unit. Consequently, the same timing is valid and the vector start-up time is not affected. Only the peak performance increases accordingly.

The execution time of the kernels $T(v_{\text{kernel}})$ satisfies

$$T\left(v_{\text{kernel}}\right) = T_{\text{start-up}} + 2kv_{\text{kernel}}\frac{\tau_{\text{proc}}}{n_{\text{pipe}}}, \text{ for } v_{\text{kernel}} \geq 2v_l. \quad (10)$$

For the single vector operations, such as the square norm, the inner product, and the saxpy kernel, k equals 1. The vector start-up time $T_{\text{start-up}}$ depends upon which kernel is performed. In Table 2, the vector start-up times are given for the various kernels.

For $v_{\text{kernel}} \gg 1$, the number of floating-point operations for each kernel $Ops(v_{\text{kernel}})$ approximately satisfies

$$Ops(v_{\text{kernel}}) = 2kv_{\text{kernel}}. \quad (11)$$

The vector start-up times for a single and a multiple saxpy are identical. Therefore, if the effective vector length v of the kernels is defined as

TABLE 2
THE KERNELS AND THEIR VECTOR START-UP TIMES

Kernel	$T_{\text{start-up}}$
$\lVert \underline{x} \rVert^2$	$2\tau_{\text{mem}} + \tau_{\text{mult}} + v_l\tau_{\text{add}} + (3v_l - 2)\,\tau_{\text{proc}}$
$(\underline{x}, \underline{y})$	$2\tau_{\text{mem}} + \tau_{\text{mult}} + v_l\tau_{\text{add}} + (3v_l - 2)\,\tau_{\text{proc}}$
$\underline{y}' = \underline{y} + \alpha\underline{x}$	$2\tau_{\text{mem}} + \tau_{\text{mult}} + \tau_{\text{add}} - \tau_{\text{proc}}$
$\underline{y}' = \underline{y} + \sum_{j=1}^{k} \alpha_j \underline{x}_j$	$2\tau_{\text{mem}} + \tau_{\text{mult}} + \tau_{\text{add}} - \tau_{\text{proc}}$

$$v = kv_{\text{kernel}}, \qquad (12)$$

the parameter k can be left out. As each processor executes exactly one kernel, the total number of operations that are executed by all processors satisfies $pOps(v)$. Hence, the performance as function of the vector length v on p processors $P(p, v)$ is

$$P(p, v) = \frac{pOps(v)}{T(v)} = \frac{1}{\frac{1}{P(p)} + \frac{T_{\text{start-up}}}{2pv}}. \qquad (13)$$

The ratio of the sustainable and the theoretical peak performance is

$$\frac{P(p, v)}{P(p)} = \frac{1}{1 + \frac{1}{2}P(1)\frac{T_{\text{start-up}}}{v}}. \qquad (14)$$

An important characteristic of vector processing is due to [14]. Since the performance depends on the vector length, the half performance vector length $v_{1/2}$ quantifies the performance behavior. The smaller $v_{1/2}$, the faster the theoretical peak performance is approached. The vector length for $P(p, v_{1/2}) = \frac{1}{2}P(p)$ is

$$v_{1/2} = \frac{1}{2}P(1)T_{\text{start-up}}. \qquad (15)$$

Due to the lack of interprocessor communication, $v_{1/2}$ is independent of p.

5 SHARED MEMORY

So far, the memory-access time τ_{mem} has not been considered in detail. Besides the memory-chip access time, the interconnection network causes additional delay that contributes to the memory-access time. Because the additional delay depends on the size of the network, and this size depends on the number of processors that are connected to the shared memory, the memory-access time depends on the number of processors.

Under the assumption that there are no memory bank conflicts, all processors access different memory banks at all times. As there are multiple memory banks connected to a single multiplexer, it may happen that multiple processors want to access the same multiplexer in order to access different banks that are connected with this multiplexer. This causes conflicts. If it is assumed that no such conflicts arise, the interconnection network must permit every connection permutation between the pathways and the multiplexers. In this way the shared memory exhibits a uniform memory access (UMA).

Many different interconnection networks are known. Networks that allow every connection permutation without conflicts are nonblocking, and rearrangeable nonblocking networks. Such networks were proposed by Clos [5] and Benes [4], respectively. Yeh and Feng [28] describe a class of networks that are rearrangeable nonblocking. Other interconnection networks, such as the perfect shuffle network [25], the omega network [17], the baseline network [27], and the delta network [23], for example, allow only a subclass of connection permutations. The simplest interconnection network is a crossbar switch that can perform all connection permutations between the pathways and the multiplexers without conflict.

The total number of connections between the memory banks and the processors is pn_{path}. If the number of inputs and outputs of a crossbar switch is equal to b and $pn_{\text{path}} \le b$, the processor-memory network can be a single $b \times b$ crossbar switch. The connections are bidirectional, i.e., the inputs and outputs can interchange, corresponding to read and write accesses of the shared memory. Because the circuit complexity of a crossbar switch is $O(b^2)$, the switch size is limited. Typically, b is ≈ 16 [15]. For a single crossbar switch network, the memory-access time is

$$\tau_{\text{mem}} = \tau_{\text{chip}} + \tau_{\text{crossbar}}. \qquad (16)$$

with τ_{chip} the memory-chip access time and τ_{crossbar} the delay introduced by the crossbar switch. The delay caused by the memory bank multiplexers is ignored. Consequently, if $pn_{\text{path}} \le b$, the memory-access time does not dependent upon p.

If $pn_{\text{path}} > b$, it is not possible to interconnect all pathways to all memory bank multiplexers with a single crossbar switch. To remove this limitation, multiple crossbar switches can be cascaded. Obviously, such a multistage interconnection network (MIN) increases the memory-access time. The rearrangeable MINs of [28] require $2\log_b(pn_{\text{path}})$ stages of $b \times b$ crossbar switches, increasing the memory-access time to

$$\tau_{\text{mem}} = \tau_{\text{chip}} + 2\tau_{\text{crossbar}}\log_b(pn_{\text{path}}). \qquad (17)$$

Note that additional overhead caused by routing has not been taken into account. Routing, i.e., setting up of switches in the MIN for a given permutation, is not straightforward. The parallel algorithms of [18], [21], and [28] to determine the switch settings require $O(\log_b^2(pn_{\text{path}}))$ time. Therefore, for arbitrary permutations, routing dominates the terms in (17). However, for important subclasses of permutations, [20] and [24] propose self-routing algorithms that set the switches as data passes through the MIN. Consequently, (17) is a lower bound.

With (17), the memory bandwidth satisfies

$$B = \frac{m}{\tau_{\text{chip}} + 2\tau_{\text{crossbar}}\log_b\left(pn_{\text{path}}\right)}, \qquad (18)$$

and the memory bandwidth inequality (7) becomes

$$pR\frac{n_{\text{pipe}}}{\tau_{\text{proc}}} \le \frac{m}{\tau_{\text{chip}} + 2\tau_{\text{crossbar}}\log_b\left(pn_{\text{path}}\right)}. \qquad (19)$$

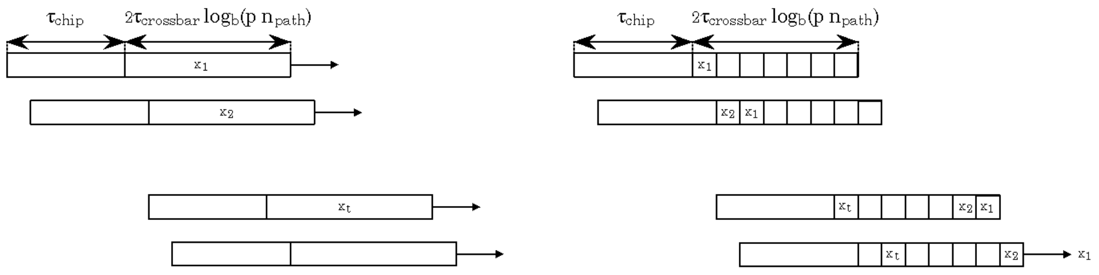

Fig. 5. Consecutive memory bank read access with an unbuffered and a buffered MIN. In the first case, a connection through all stages is reserved for the complete transfer time, In the latter, a connection through only one stage is reserved each time.

For $pn_{path} \gg 1$, the memory-chip access time can be neglected and the maximum number of processors running at peak performance is bounded by

$$pn_{pipe}\log_b\left(pn_{path}\right) < \frac{1}{R}\frac{\tau_{proc}}{\tau_{crossbar}}\frac{m}{2}. \quad (20)$$

In order to guarantee computation bound processing (cf. (3)), the number of pathways must always satisfy

$$n_{path} \geq Rn_{pipe}. \quad (21)$$

Minimizing the number of stages, implies a lower bound on n_{path}, so (20) changes to

$$pn_{pipe}\log_b\left(pn_{pipe}R\right) < \frac{1}{R}\frac{\tau_{proc}}{\tau_{crossbar}}\frac{m}{2}. \quad (22)$$

Although it is more likely that $\tau_{crossbar}$ will increase with the switch size b, it is assumed that there exists a fixed lower bound on $\tau_{crossbar}$. As soon as this limit has been attained, the number of processors and/or the number of pipelined functional units per processor and, thus, the peak performance can only further be improved by increasing the number of memory banks. Doubling the number of processors and the memory size at the same time, implies that the number of memory banks must more than double to attain the peak performance. This hampers scalability.

However, this bottleneck can be resolved. Up to now, the MIN was assumed to be indivisible. A single transfer occupies one switch at every stage during the complete transfer time. Consequently, all switches that are involved are occupied and the MIN latency increases the memory-access time. In the literature, this is referred to as circuit switching [10]. Because the MIN consists of several stages, it is possible to apply buffering at the intermediate stages. If, at each stage, the switches are equipped with simple latches, a transfer through the network can be pipelined, and the delay caused by each stage can be overlapped with all other stages. This form of packet switching [10] effectively improves the MIN throughput. The already mentioned self-routing algorithms can efficiently be used in this configuration. However, the total memory-access time for a single memory access does not improve (see Figs. 5a and 5b).

Note that this simple form of buffering is adequate only, when all transfers are unidirectional accesses. For bidirectional accesses, read-write packet collisions might occur. Consequently, conflict resolution must be incorporated to permit redundant operation of the MIN. A way to avoid such collisions might be to split up the network in two parts: one part that handles the read accesses and another part that takes care of the write accesses. The two networks can operate in parallel, thus reducing the number of interconnections for each network. Still, for $b = 2$, this reduces the number of connections by at most a factor of two, so, effectively, at most one MIN stage less is required. Furthermore, an additional stage of switches is required at the bank multiplexer side to guarantee that both networks can access all memory banks. Another issue not discussed here concerns the actual cycle time of the network. The processor cycle time τ_{proc} is assumed to be a multiple of the MIN cycle time.

If, for simplicity, only unidirectional accesses are considered, the MIN acts as a delay line. Consequently, the memory bandwidth is independent of p and it satisfies

$$B = \frac{m}{\tau_{chip}}. \quad (23)$$

Hence, the doubling of the number of processors and memory banks at the same time no longer penalizes sustainable peak performance (cf. (18)).

As the memory-access time still satisfies (17), the vector start-up time for a kernel is

$$T_{start\text{-}up} = 2\tau_{mem} + f\left(\tau_{mult}, \tau_{add}, \tau_{proc}, v_l\right), \quad (24)$$

$$= 4\tau_{crossbar}\log_b\left(pn_{path}\right) + 2\tau_{chip} + f\left(\tau_{mult}, \tau_{add}, \tau_{proc}, v_l\right). \quad (25)$$

The function $f(\tau_{mult},\tau_{add},\tau_{proc},v_l)$ represents the kernel dependent parameters. The term $2\tau_{mem}$ is the same for every kernel (see Table 2). In fact, other kernels not considered here exhibit the same behavior; at least one nonoverlapped read and one nonoverlapped write access is required for *any* kernel. For large p, only the first term of the r.h.s. becomes significant; all other terms are independent of p. Therefore, scheduling and vector register allocation are of minor importance as long as all functional units can operate concurrently without conflict. When $T_{start\text{-}up}$ depends on p, $v_{1/2}$ also depends on p. With the lower bound of (21), and $pn_{pipe} \gg 1$, (15) changes to

$$v_{1/2} > 2\frac{\tau_{crossbar}}{\tau_{proc}}n_{pipe}\log_b\left(pn_{pipe}R\right). \quad (26)$$

A reduction of a factor of 1/2 can be realized in (26) by sacrificing rearrangeability of the MIN. In this case, only $\log_b(pn_{path})$ network stages are required, and self-routing algorithms exist which do not cause additional terms in (26) (see [17]). However, due to the fact that conflicts might arise in the MIN, conflict resolution must be incorporated for a redundant network operation. Buffered MINs that are not rearrangeable can be characterized by the model, the buffer size at each switch and the control signal propagation. Dias and Jump [6] and Mun and Youn [19] propose models for buffered delta and Banyan networks, respectively. For so-called small clock cycle designs, where the control signal propagation is limited to only one network stage, [7] and [29] propose models for buffered delta and Banyan networks, respectively. In these models, the buffers are operating with a FIFO arbitration. Tamir and Chi [26] discuss the design of a crossbar switch that incorporates buffers with non-FIFO arbitration to resolve conflicts. They also simulate an omega network with this buffered crossbar switch.

The implications of (26) are serious. To proportionally increase the performance with p, n_{pipe}, and R remaining fixed, the vector length of the kernels must increase with $O(\log_b p)$ and the total amount of floating-point operations must increase with $O(p \log_b p)$. For ideal algorithms under ideal conditions, this last observation is truly distressing.

6 Performance Behavior of the LINPACK Benchmark

As the model was derived under the assumption of ideal conditions, its scope might be considered academic. However, in special cases where the actual circumstances approach the ideal, it is possible to accurately predict the performance behavior of parallel vector computers with a shared memory for a specific algorithm.

In the LINPACK benchmark, large linear systems are solved with LU decomposition with partial pivoting [8]. For a system with n unknowns, $2n^3/3 + 2n^2$ floating-point operations are required. Although there exist other methods which involve block matrix operations [11], here only a straightforward implementation of LU decomposition is considered. The dominant part of the computations is the saxpy with an average vector length of $2n/3$ [9]. Partial pivoting requires $n^2/2$ comparisons and n indirect addressing operations. Furthermore, $n^2/2$ floating-point divisions are required. If communication overheads are neglected and it is assumed that the total number of operations is distributed equally over all pipelined floating-point units, then, for large n, the effects of partial pivoting and the divisions can be neglected because the amount of operations to be processed by a single pipelined functional unit is $(2n^3/3 + 2n^2)/(pn_{\text{pipe}})$ with $pn_{\text{pipe}} \ll n$.

Given the average vector length, it is natural to wonder whether (14) also holds for the average vector length. It is easy to show that this conjecture is correct. If each of the p processors executes a sequence of saxpys with consecutive vector lengths $v_1, v_2, \ldots, v_q$, the total number of operations performed is $p\sum_{i=1}^{q} 2v_i$ and the execution time is

$$\sum_{i=1}^{q}\left\{T_{\text{start-up}} + 2v_i \frac{\tau_{\text{proc}}}{n_{\text{pipe}}}\right\}$$

(cf. (10)-(12)). The overall performance satisfies

$$\frac{p\sum_{i=1}^{q} 2v_i}{\sum_{i=1}^{q} T_{\text{start-up}} + \frac{\tau_{\text{proc}}}{n_{\text{pipe}}}\sum_{i=1}^{q} 2v_i} = \frac{p}{\frac{\tau_{\text{proc}}}{n_{\text{pipe}}} + \frac{qT_{\text{start-up}}}{\sum_{i=1}^{q} 2v_i}} = \frac{p}{\frac{1}{P(1)} + \frac{1}{2}\frac{T_{\text{start-up}}}{\frac{1}{q}\sum_{i=1}^{q} v_i}}. \quad (27)$$

Hence, the ratio w.r.t. the theoretical peak performance is

$$\frac{1}{1 + \frac{1}{2}P(1)\frac{T_{\text{start-up}}}{\overline{v}}} = \frac{P(p, \overline{v})}{P(p)}, \text{ with } \overline{v} = \frac{1}{q}\sum_{i=1}^{q} v_i. \quad (28)$$

If each processor executes an equal fraction of saxpys with the same average vector length $\overline{v} = \frac{2}{3}n$, then the performance ratio as function of the linear system size n is

$$\frac{P(p,n)}{P(p)} = \frac{1}{1 + \frac{3}{4}P(1)\frac{T_{\text{start-up}}}{n}}. \quad (29)$$

If $n_{1/2}$ is the size to reach half the theoretical peak performance, the ratio becomes

$$\frac{P(p,n)}{P(p)} = \frac{1}{1 + \frac{n_{1/2}}{n}}, \text{ with } n_{1/2} = \frac{3}{4}P(1)T_{\text{start-up}}. \quad (30)$$

Equation (30) gives the upper bound that can only be approximated when n and $n_{1/2}$ are large. Based on the fact that the number of operations and the execution time can both be represented as polynomials of the same degree in n, Arnold [1] derives an equation for the LINPACK performance that appears to be similar. However, since not all the polynomial coefficients are known, he is unable to show that (30) is an absolute upper bound with $P(p)$ equal to the theoretical peak performance, and he cannot determine $n_{1/2}$. Consequently, a two-parameter least-squares fit of performance data is inappropriate. With (25) and the lower bound of n_{path} in (21), $n_{1/2}$ is

$$n_{1/2} = \frac{3}{4}\frac{n_{\text{pipe}}}{\tau_{\text{proc}}}\left\{4\tau_{\text{crossbar}} \log_b\left(pn_{\text{pipe}}R\right) + 2\tau_{\text{chip}} + f\left(\tau_{\text{mult}}, \tau_{\text{add}}, \tau_{\text{proc}}, v_l\right)\right\}. \quad (31)$$

Consequently, $n_{1/2}$ increases with $O(\log_b p)$. Due to the fact that scheduling and vector register allocation of real parallel vector computers are likely to be different from the presumed machine characteristics mentioned in Section 4, the actual implementation of the saxpy kernel might be different. Because the function f depends on the characteristics of the subsequent machine, and data such as τ_{add} and τ_{mult} are not available—such data is considered proprietary—f cannot be estimated. Still, however, with (30), it is possible to predict the performance behavior.

In Table 3, the measured maximal performance of the LINPACK benchmark $P_m(n_{\max})$ is compared with the performance predicted by (30), given $n_{1/2}$ and the maximal size of the linear system $n_{\max}$. Although [8] defines $n_{1/2}$ as the size of the linear system to reach half the maximal performance, instead of half the theoretical peak performance, the predicted performance according to (30) closely matches the measurements. The relative error is less than 10 percent

TABLE 3
LINPACK PERFORMANCE OF PARALLEL VECTOR COMPUTERS WITH A SHARED MEMORY (DERIVED FROM [8, TABLE 3])

Computer	$n_{1/2}$	$n_{\max}$	$P(p)$ Gflop/s	$P_m(n_{\max})$ Gflop/s	$P(p, n_{\max})$ Gflop/s	Error %
Hitachi S-3800/480	830	15,500	32	28.4	30.4	7.0
NEC SX-3/44R	830	6,400	26	23.2	23.0	–0.9
Hitachi S-3800/380	760	15,680	24	21.6	22.9	6.0
NEC SX=3/44	832	6,144	22	20.0	19.4	–3.0
NEC SX-3/34R	691	6,144	19	17.4	17.1	–1.7
Hitachi S-3800/280	570	15,680	16	14.6	15.4	5.5
Cray Y-MP C90	650	10,000	15	13.7	14.1	2.9
NEC SX-3/42R	516	4,352	13	11.6	11.6	0.0
NEX SX-3/24R	492	4,352	13	11.6	11.7	0.9
NEC SX-3/24	500	4,352	11	10.0	9.9	–1.0
NEC SX-3/42	640	4,608	11	10.0	9.7	–3.0
NEC SX-3/32R	717	6,144	9.6	8.7	8.6	–1.1
Hitachi S-3800/180	470	15,680	8	7.4	7.8	5.4
Cray J932	550	10,000	6.4	5.8	6.1	5.2
NEC SX-3/41R	414	3,584	6.4	5.8	5.7	–1.7
NEC SX-3/22R	370	3,072	6.4	5.8	5.7	–1.7
NEC SX-3/14R	282	2,816	6.4	5.8	5.8	0.0
NEC SX-3/22	384	3,072	5.5	5.0	4.9	–2.0
NEC SX-3/14	384	3,072	5.5	5.0	4.9	–2.0
NEC SX-3/31R	414	6,144	4.8	4.4	4.5	2.3
NEC SX-3/21R	257	2,560	3.2	2.9	2.9	0.0
NEC SX-3/12R	174	2,048	3.2	2.9	2.9	0.0
Cray J916	360	10,000	3.2	2.8	3.1	10.7
NEC SX-3/12	256	2,048	2.8	2.5	2.5	0.0
NEC SX-3/11R	130	2,048	1.6	1.5	1.5	0.0
NEC SX-3/11	192	2,816	1.4	1.3	1.3	0.0
NEC SX-3/1LR	112	2,304	.8	.78	.76	–2.6
NEC SX-3/1L	128	2,084	.68	.67	.64	–4.5

The measured and the predicted performance $P_m(n_{\max})$ and $P(p, n_{\max})$ resp. are compared. The error is determined w.r.t. $P_m(n_{\max})$.

in all but one case. As (30) gives the absolute upper bound, the relative error cannot become negative. An explanation might be the following: If, due to a small system size, the effects of pivoting, the divisions and the communication overhead slightly increase the execution time, $n_{1/2}$ must increase to reach half the theoretical peak performance. Consequently, the performance ratio for $n_{\max}$ is underestimated, which accounts for negative relative errors.

Table 4 shows the performances of the top 10 fastest distributed memory machines. It is remarkable that (30) also gives an accurate prediction for distributed memory machines. Apparently, for large n, the communication overhead becomes independent of n and an additional term, which is constant, can account for it in (24).

7 CONCLUSIONS

Although only the dot, square norm, and saxpy kernels were considered, similar derivations can be given for other kernels. Due to the fact that the dominant term in the vector start-up time (see Table 2) is $2\tau_{\text{mem}}$, corresponding to a nonoverlapped read and write access of the shared memory, other kernels behave in a similar manner. For every kernel, at least one read and one write is necessary.

In order to achieve a uniform memory access (UMA), every pathway should be able to access every memory bank. For a conflict free interconnection network operation, the MIN must be rearrangeable and, based on the rearrangeable MINs derived by [28], $2\log_b(pn_{\text{path}})$ stages are required. For general connection permutations routing overhead is $O(\log_b^2(pn_{\text{path}}))$, which dominates the MIN latency of $O(\log_b(pn_{\text{path}}))$. However, for important subclasses of connection permutations, self-routing algorithms exist so that routing overhead can be neglected. If the MIN is unbuffered, the maximal memory bandwidth is determined by the number of memory banks, the memory-chip access time, and the MIN latency. In this case, the theoretical peak performance can only be achieved if the number of memory banks increases with $O(p \log_b p)$. If the MIN is buffered, pipelining can be applied and the maximal memory bandwidth becomes independent of the MIN latency. The maximum achievable performance is proportional to the number of memory banks. The number of stages can be reduced to $\log_b(pn_{\text{path}})$ by sacrificing the rearrangeability. In this case, not all connection permutations can be realized without conflict and conflict resolution with buffering must be incorporated. This reduces the memory-access time at the

TABLE 4
LINPACK PERFORMANCE OF THE TOP 10 FASTEST DISTRIBUTED MEMORY MACHINES
(DERIVED FROM [8, TABLE 3])

Computer	p	$n_{1/2}$	$n_{\max}$	$P(p)$ Gflop/s	$P_m(n_{\max})$ Gflop/s	$P(p, n_{\max})$ Gflop/s	Error %
Intel Paragon XP/S MP	6,768	25,700	128,600	338	281.1	281.7	0.2
Intel Paragon XP/S MP	6,144	24,300	122,500	307	256.2	256.2	0.0
Intel Paragon XP/S MP	5,376	22,900	114,500	269	223.6	224.2	0.3
Intel Paragon XP/S MP	4,608	21,000	106,000	230	191.5	192.0	0.3
Numerical Wind Tunnel	140	13,800	42,000	236	170.4	177.6	4.2
Numerical Wind Tunner	128	13,120	40.960	216	157.9	163.6	3.6
Intel Paragon XP//S MP	3,648	18,100	95,000	182	151.7	152.9	0.8
Fujitsu VPP500/128	128	13,120	40,960	205	149.7	155.3	3.7
Intel Paragon XPS-140	3,680	20,500	55,700	184	143.4	134.5	–6.2
Paragon XP/S MP	3,072	17,800	86,000	154	127.1	127.6	0.4

The measured and the predicted performance $P_m(n_{\max})$ and $P(p, n_{\max})$ resp. are compared. The error is determined w.r.t. $P_m(n_{\max})$.

TABLE 5
SUMMARY OF SYMBOLS

Symbol	Description
b	number of in- and outputs of a crossbar switch
B	memory bandwidth
B_{req}	required memory bandwidth to sustain peak performance
$f()$	part of $T_{\text{start-up}}$ which is independent of the memory-access time
m	number of memory banks
$n_{\max}$	maximal linear system size for a specified machine
n_{path}	number of pathways between a processor and the shared memory
n_{pipe}	number of pipelined floating-point units per processor
$n_{1/2}$	linear system size to reach half the theoretical peak performance
$Ops(v)$	number of floating-point operations for a single kernel with a vector length v
p	number of processors
$P_{\max}$	maximal sustainable peak performance based on throughput
$P_m(n_{\max})$	measured LINPACK performance for a system size $n_{\max}$ on a specified machine
$P(p)$	theoretical peak performance of p processors
$P(p, n)$	sustainable peak performance of p processors for a linear system size n
$P(p, v)$	sustainable peak performance of p processors for kernels with a vector length v
R	reference to floating-point operations ratio
$T_{\text{start-up}}$	vector start-up time of a kernel
$T(v)$	execution time as function of the vector length v
τ_{add}	addition time
τ_{chip}	memory-chip access time
τ_{crossbar}	crossbar switch delay
τ_{mem}	memory-access time
τ_{mult}	multiplication time
τ_{proc}	processor cycle time
v	effective vector length
$\bar{v}$	average vector length
v_{kernel}	vector length of a kernel
v_l	maximal number of elements in a vector register
$v_{1/2}$	vector length to reach half the theoretical peak performance

expense of possible MIN conflicts. In all cases, the memory-access time grows at least as fast as $O(\log_b p)$. Therefore, if the performance should scale up linearly with p, the vector length of any kernel must increase with $O(\log_b p)$. This amounts to $O(p \log_b p)$ floating-point operations. Evidently, it is not just the memory bandwidth that is the main characteristic of a shared memory; the memory latency also affects the performance behavior. The performance measurements of LINPACK concurred with the performance prediction based on the model. The half performance system size increases with $O(\log_b p)$. If the memory-access pattern is less regular, the memory bandwidth decreases due to memory bank conflicts and, possibly, interconnection network conflicts, and it affects the achievable performance significantly. Therefore, the shared memory architecture is not very scalable.

Apparently, a single shared memory based on a uniform memory access is too costly. As parallel algorithms exhibit topological structure, some form of data locality always exists. In order to exploit such locality, a single shared memory is not needed, since it is not required that all data is to be shared by all processors all the time. On the other hand, given the topological structures of an algorithm and a distributed memory architecture, the performance heavily depends on the fit of both. This issue causes portability problems. For shared memory architectures, the topological structure of algorithms is not important and portability is much better. With the advent of vector processing, new algorithms and techniques had to be developed to exploit performance characteristics of pipelined functional units. At this moment, these techniques have matured so that significant benefits can be achieved on parallel vector computers. Similar to vector processing, the introduction of distributed memory machines has caused a serious trend change. Therefore, it will be clear that a certain time must pass before new algorithms and techniques emerge so that distributed memory machines can effectively compete with shared memory parallel vector computers on general applications.

ACKNOWLEDGMENTS

The author would like to thank the referees for their valuable comments.

REFERENCES

[1] C.N. Arnold, "Methods for Performance Evaluation of Algorithms and Computers," *Computers in Physics*, vol. 4, no. 5, pp. 514-520, Sept. / Oct. 1990.

[2] G. Bilardi and F.P. Preparata, "Horizons of Parallel Computation," *J. Parallel and Distributed Computing*, vol. 27, no. 2, pp. 172-182, June 1995.

[3] G. Bell, "Ultracomputers: A Teraflop Before Its Time," *Comm. ACM*, vol. 35, no. 8, pp. 27-47, Aug. 1992.

[4] V.E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic Press, 1965.

[5] C. Clos, "A Sudy of Non-Blocking Switching Networks," *Bell System Technical J.*, vol. 32, pp. 406-424, Mar. 1953.

[6] D.M. Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Networks," *IEEE Trans. Computers*, vol. 30, no. 4, pp. 273-282, Apr. 1981.

[7] J. Ding and L.N. Bhuyan, "Finite Buffer Analysis of Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 43, no. 2, pp. 243-247, Feb. 1994.

[8] J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," Technical Report CS-89-85, Univ. of Tennessee and Oak Ridge Nat'l Laboratory, Nov. 1995.

[9] J.J. Dongarra, "The LINPACK Benchmark: An Explanation," *Lecture Notes in Computer Science*, vol. 297, pp. 456-474. Berlin: Springer, 1988.

[10] T.-Y. Feng, "A Survey of Interconnection Networks," *Computer*, vol. 14, no. 12, pp. 12-27, Dec. 1981.

[11] G.H. Golub and C.F. Van Loan, *Matrix Computations*, second ed., chapter 3. Baltimore: The Johns Hopkins Univ. Press, 1989.

[12] J.J. Hack, "Peak vs. Sustained Performance in Highly Concurrent Vector Machines," *Computer*, vol. 19, no. 9, pp. 11-19, Sept. 1986.

[13] M.D. Hill, "What Is Scalability?" *Computer Architecture News*, vol. 18, no. 4, pp. 18-21, Dec. 1990.

[14] R.W. Hockney, "Super-Computer Architecture," *Infotech State of the Art Report: Future Systems 2*, pp. 277-305. Maidenhead: Infotech, 1977.

[15] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, p. 338. New York: McGraw-Hill, 1993.

[16] V. Kumar and A. Gupta, "Analysis of Scalability of Parallel Algorithms and Architectures: A Survey," *Proc. Int'l Conf. Supercomputing*, pp. 396-405, 1991.

[17] D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, vol. 24, no. 12, pp. 1,145-1,155, Dec. 1975.

[18] G.F. Lev, N. Pippenger, and L.G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Trans. Computers*, vol. 30, no. 2, pp. 93-100, Feb. 1981.

[19] Y. Mun and H.Y. Youn, "Performance Analysis of Finite Buffered Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 43, no. 2, pp. 153-162, Feb. 1994.

[20] D. Nassimi and S. Sahni, "A Self-Routing Benes Network and Parallel Permutation Algorithms," *IEEE Trans. Computers*, vol. 30, no. 5, pp. 332-340, May 1981.

[21] D. Nassimi and S. Sahni, "Parallel Algorithms to Set-Up the Benes Permutation Network," *Proc. Workshop Interconnection Networks for Parallel and Distributed Processing*, pp. 70-71, 1980.

[22] D. Nusbaum and A. Agarwal, "Scalability of Parallel Machines," *Comm. ACM*, vol. 34, no. 3, pp. 56-61, Mar. 1991.

[23] J.H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Trans. Computers*, vol. 30, no. 10, pp. 771-780, Oct. 1981.

[24] C.S. Raghavendra and R.V. Boppana, "On Self-Routing in Benes and Shuffle-Exchange Networks," *IEEE Trans. Computers*, vol. 40, no. 9, pp. 1,057-1,064, Sept. 1991.

[25] H.S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Computers*, vol. 20, no. 2, pp. 153-161, Feb. 1971.

[26] Y. Tamir and H.-C. Chi, "Symmetric Crossbar Arbiters for VLSI Communication Switches," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 1, pp. 13-27, Jan. 1993.

[27] C.-L. Wu and T.-Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 29, no. 8, pp. 694-702, Aug. 1980.

[28] Y.-M. Yeh and T.-Y. Feng, "On a Class of Rearrangeable Networks," *IEEE Trans. Computers*, vol. 41, no. 11, pp. 1,361-1,379, Nov. 1992.

[29] H.Y. Youn and Y. Mun, "On Multistage Interconnection Networks with Small Clock Cycles," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 1, pp. 86-93, Jan. 1995.

Eskil Dekker received the MS degree in applied physics from the Delft University of Technology, The Netherlands, in 1987, and the PhD degree in technical sciences from the same university in 1995. From 1987 to 1990, he was with CERFACS in Toulouse, France, investigating algorithms and architectures for parallel computing. Currently, he is a postdoctoral fellow with the Faculty of Information Technology and Systems of the Delft University of Technology. Dr. Dekker is a member of the IEEE, the ACM, and EUROSIM. His research interests are in high-performance computing, specifically in fast numerical algorithms, highly parallel computer architectures, and scalability.