CMU-CS-85-158

Memory Requirements for Balanced Computer Architectures

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

June 1985

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contract N00014-80-C-0236, NR 048-659.

Abstract

In this paper, a processing element (PE) is characterized by its computation bandwidth, I/O bandwidth, and the size of its local memory. In carrying out a computation, a PE is said to be *balanced* if the computing time equals the I/O time. Consider a balanced PE for some computation. Suppose that the computation bandwidth of the PE is increased by a factor of α relative to its I/O bandwidth. Then when carrying out the same computation the PE will be imbalanced, i.e., it will have to wait for I/O. A standard method to avoid this I/O bottleneck is to reduce the overall I/O requirement of the PE by increasing the size of its local memory. This paper addresses the question of by how much the PE's local memory must be enlarged in order to restore balance.

The following results are shown: For matrix computations such as matrix multiplication and Gaussian elimination, the size of the local memory must be increased by a factor of α^2. For computations such as relaxation on a k-dimensional grid, the local memory must be enlarged by a factor of α^k. For some other computations such as the FFT and sorting, the increase is exponential, i.e., the size of the new memory must be the size of the original memory to the α-th power. All these results indicate that to design a balanced PE, the size of its local memory must be increased much more rapidly than its computation bandwidth. This phenomenon seems to be common for many computations where an output may depend on a large subset of the inputs.

Implications of these results for some parallel computer architectures are discussed. One particular result is that to balance an array of p linearly connected PEs for performing matrix computations such as matrix multiplication and matrix triangularization, the size of each PE's local memory must grow linearly with p. Thus, the larger the array is, the larger each PE's local memory must be.

1. INTRODUCTION

With today's technology, the challenge in designing a high-performance computer is usually not in providing processing elements with the required high computation bandwidths, but in making sure that information can flow to and from these elements with sufficient speed. For example, very fast processing elements can be built using off-the-shelf 16 MHz, 32-bit microprocessors [5] and/or floating-point chips capable of delivering 10 million operations per second [2]. The computation bandwidth of such a processing element can be further increased by incorporating multiple copies of these chips and operating them in parallel. However, the I/O bandwidth with the rest of the system (e.g., system memory and interconnections) cannot be increased as easily, and as a result it often becomes a bottleneck for the performance of the entire system.

A standard approach to alleviating this I/O problem is to provide a local memory at a processing element. This local memory can "cache" frequently used data and instructions, so that the required I/O bandwidth with the outside world is reduced. It is well-known that the size of the local memory must be large if the computation bandwidth of the processing element is large, as represented by the "Amdahl's rule" [8]. But exactly how large should this local memory be? This paper answers the question for several important computational tasks.

To help study the problem formally, an information model is introduced in Section 2 to characterize a processing element. Section 3 derives results on how the local memory of a processing element must be increased as the computation bandwidth increases. Section 4 discusses implications of these results for some parallel computer architectures. Concluding remarks are provided in Section 5.

2. THE INFORMATION MODEL

As illustrated in Figure 1, we characterize a processing element (PE) by:

1. *C: the computation bandwidth*, which is the number of operations that the PE can deliver per second,

2. *IO: the I/O bandwidth*, which is the number of words that the PE can communicate with the outside world per second, and

3. *M: the size of the PE's local memory*, in terms of number of words.

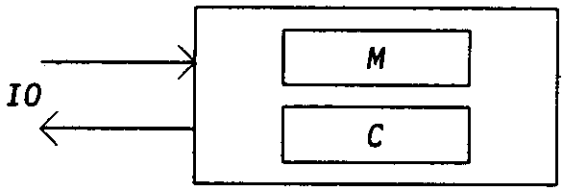

Figure 1. Processing element characterized by its computation bandwidth (C), I/O bandwidth (IO), and size of local memory (M).

In carrying out a computation such as the fast Fourier transform (FFT) or matrix multiplication, a PE is said to be *balanced* if the I/O time equals the computing time. When a PE is balanced for a given computation, we know that its computation, I/O and memory subsystems are not over- or under-designed for that computation. A challenge for computer architects is to keep a PE balanced, while taking advantage of technological opportunities such as large increases in computation bandwidth. Since it is usually difficult or expensive to increase the I/O bandwidth, we ask the following question:

> Assume that a PE is balanced for a given computation. Now C/IO is increased by a factor of α. To re-balance the PE for the same computation (without increasing IO), by how much must M be increased?

The following symbols and equalities are useful in deriving answers to the question. For carrying out a given computation on a PE, let C_{comp} ("cost for computation") and C_{io} ("cost for I/O") denote the total number of operations needed for the computation and for the I/O, respectively. We assume that one I/O operation can transfer a word to or from the PE. Then the computing and I/O times

are C_{comp}/C and C_{io}/IO, respectively. Therefore, the PE is balanced if and only if

$$\frac{C_{comp}}{C} = \frac{C_{io}}{IO},$$

or

$$\frac{C}{IO} = \frac{C_{comp}}{C_{io}}. \tag{1}$$

Now suppose that C/IO is increased by a factor of α. Then by (1) *the PE is re-balanced if and only if the ratio C_{comp}/C_{io} is increased by a factor of α*. This provides a method that we can use in re-balancing a PE. For many computations, this can be accomplished by increasing the size of the PE's local memory.

To be precise, let M_{old} be the size of the original local memory, and M_{new} the *minimum* size of the new memory necessary to re-balance the PE. In the rest of the paper, we study by how much (expressed in terms of α) M_{new} must be larger than M_{old}.

3. RESULTS FOR SOME COMPUTATIONS

Consider a PE that is balanced for a given computation. Now suppose that C/IO is increased by a factor of α. This section drives answers to the question proposed in the preceding section for several computations. The following is a summary of the results:

- Matrix computation such as matrix multiplication and triangularization: $M_{new} = \alpha^2 M_{old}$;
- Grid computation:
  - 2-dimensional: $M_{new} = \alpha^2 M_{old}$;
  - d-dimensional: $M_{new} = \alpha^d M_{old}$;
- FFT: $M_{new} = (M_{old})^{\alpha}$;
- Sorting: $M_{new} = (M_{old})^{\alpha}$;
- I/O bounded computations such as matrix-vector multiplication and solution of triangular linear systems: Impossible, i.e., PE cannot be re-balanced by merely enlarging its local memory, without increasing its IO bandwidth.

Throughout this section, we will assume that for all the computations the problem size N is arbitrarily large, and that N is much larger than the size of the PE's local memory M.

3.1. Matrix Multiplication

Consider the problem of multiplying two $N \times N$ matrices, assuming a local memory of size M. In the following, we use a decomposition scheme that minizes the I/O requirement of the PE.

The product matrix is computed in $(N/\sqrt{M})^2$ steps, each being the computation of a $\sqrt{M} \times \sqrt{M}$ submatrix of the product matrix. Every step is a multiplication of a $\sqrt{M} \times N$ submatrix of the first input matrix with an $N \times \sqrt{M}$ submatrix of the second. This can be carried out in $C_{comp} = \Theta(N \cdot M)$ arithmetic

operations*, and $C_{io} = \Theta(N \cdot \sqrt{M})$ I/O operations. Thus,

$$\frac{C_{comp}}{C_{io}} = \Theta(\sqrt{M}). \tag{2}$$

Assume that for this computation, the PE is balanced. Now suppose that the computation bandwidth is increased by a factor of α relative to the I/O bandwidth. Then by (1), for re-balancing the PE, we must increase C_{comp}/C_{io} by a factor of α. From (2), we see that this can be done only if M is increased by a factor of α^2. That is, for this matrix multiplication computation, we have

$$M_{new} = \alpha^2 M_{old}. \tag{3}$$

The decomposition scheme we use here for matrix multiplication is just one of many possible ones. It has been shown [6] that for matrix multiplication, any decomposition scheme yields:

$$\frac{C_{comp}}{C_{io}} = h(M),$$

where the function $h(M)$ cannot exceed $\sqrt{M}$ in order of magnitude. This implies that the result of (3) is the best possible among all decomposition schemes, as far as minimizing M_{new} is concerned.

3.2. Matrix Triangularization

Given an $N \times N$ matrix A, the triangularization problem is to determine an $N \times N$ "multiplier matrix" Q and an upper triangular matrix U such that

$$QA = U.$$

By triangularization, many problems in matrix computation can be reduced to that of solving triangular linear systems. For example, this is the major step in all direct methods for solving linear systems. When M is restricted to be an orthogonal matrix, it is also the key step in computing least squares solutions and in the QR algorithm for computing eigenvalues. Gaussian elimination and Givens rotation are standard algorithms for triangularization.

The triangularization problem can be solved in $N/\sqrt{M}$ steps, where each step

---

*$f(N) = \Theta(g(N))$ means $f(N) = c \cdot g(N)$ + lower order terms in N, where c is some positive constant.

annihilates portions of $\sqrt{M}$ consecutive columns which are in the lower triangular part, and updates the rest of the matrix to prepare it for the next step. It is easy to check that the first step can be carried out in $C_{comp} = \Theta(N^2 \cdot \sqrt{M})$ arithmetic operations, and $C_{io} = \Theta(N^2)$ I/O operations, assuming a local memory of size M. Thus,

$$\frac{C_{comp}}{C_{io}} = \Theta(\sqrt{M}).$$

The same ratio is maintained for all the steps. Therefore, as in the case of matrix multiplication, we have

$$M_{new} = \alpha^2 M_{old}.$$

3.3. Grid Computation

Consider a 2-dimensional grid computation. Given an $N \times N$ grid, the task is to perform a large number of iterations on the grid, where each iteration involves updating every grid point by some weighted average of points in a surrounding window of fixed size. For some applications, on the order of N iterations may be performed. In scientific computation and image processing, this computation is usually called relaxation.

Assume that the computation is performed by an array of PEs. Each PE is responsible for the storing and updating of all the grid points in a $\sqrt{M} \times \sqrt{M}$ subgrid. For every iteration, each PE performs $C_{comp} = \Theta(\sqrt{M} \times \sqrt{M})$ arithmetic operations, and $C_{io} = \Theta(\sqrt{M})$ I/O operations. Thus, for the 2-dimensional grid computation, we have

$$M_{new} = \alpha^2 M_{old}.$$

It is straightforward to show that for a d-dimensional grid computation, we have

$$M_{new} = \alpha^d M_{old}.$$

3.4. Fast Fourier Transform

Consider the problem of computing an N-point discrete Fourier transform by the fast Fourier transform (FFT) algorithm, assuming a local memory of size M.

Decomposition for the FFT is not as straightforward as that for matrix multiplication and other computations considered above. Figure 2 depicts an N-point FFT computation and a decomposition scheme for $N=16$ and $M=4$. Results of

subcomputation blocks are shuffled before they are used as inputs of other subcomputation blocks. Note that each subcomputation block is sufficiently small so that it can be entirely carried out inside a PE with M words of local memory.

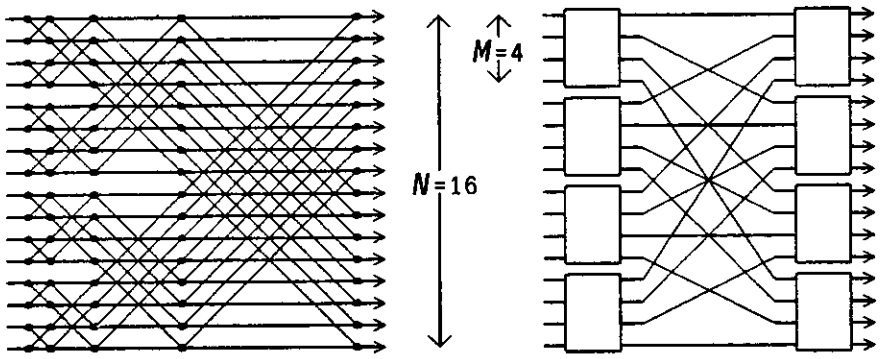

Figure 2. (a) 16-point FFT; (b) decomposing the FFT.

Each subcomputation performs $C_{comp} = \Theta(M \cdot \log_2 M)$ arithmetic operations, and $C_{io} = \Theta(M)$ I/O operations. Thus,

$$\frac{C_{comp}}{C_{io}} = \Theta(\log_2 M). \tag{4}$$

This implies that to increase the ratio C_{comp}/C_{io} by a factor of α, we must increase M to M^{α}. Therefore for the FFT, we have

$$M_{new} = (M_{old})^{\alpha}.$$

It has been shown [6] that for the FFT, any decomposition scheme yields:

$$\frac{C_{comp}}{C_{io}} = k(M),$$

where the function $k(M)$ cannot exceed $\log_2 M$ in order of magnitude. This implies that the result of (4) is the best possible among all decomposition schemes, as far as minimizing M_{new} is concerned.

3.5. Sorting

Consider the problem of sorting N keys by comparisons only. We will perform the sorting in two phases. Phase 1 sorts the N/M subsets of M keys each to produce N/M sorted lists. Phase 2 merges the sorted lists using an M-way merge algorithm. In phase 1, for each subset we perform $C_{comp} = \Theta(M \cdot \log_2 M)$ comparisons, and

$C_{io} = \Theta(M)$ I/O operations, and this can be carried out in a local memory of size M. In phase 2, for each M-way merge we maintain a heap of M elements which are the first elements of the current M sorted lists. The heap can be implemented in a memory of size M, and for each I/O operation to the heap there are $\Theta(\log_2 M)$ comparisons to be performed. Therefore for both phases, we have

$$\frac{C_{comp}}{C_{io}} = \Theta(\log_2 M).$$

Like the FFT case, this implies that for sorting,

$$M_{new} = (M_{old})^{\alpha}. \tag{5}$$

Using an information-theoretic argument, it is easy to show [9] that the result of (5) is the best possible among all sorting methods, as far as minimizing M_{new} is concerned.

3.6. I/O Bounded Computations

All the computations considered so far have been computation bounded, in the sense that computation takes more operations than I/O in order of magnitude. Computations that are not computation bounded are called *I/O bounded*. Matrix-vector multiplication and solution of triangular linear systems are examples of I/O bounded computations. For I/O bounded computations, after an increase of C/IO for a PE, there is no way to re-balance the PE by merely enlarging its local memory without increasing IO. The reason is that for these computations, inputs and intermediate results are not used more than a constant number of times on the average, so having a local memory to buffer data will not reduce the overall I/O requirement of the PE after the size of the memory exceeds certain constant.

4. IMPLICATIONS FOR SOME PARALLEL COMPUTER ARCHITECTURES

The summary of results in the beginning of Section 3 suggests a classification of computations in term of their memory requirements in achieving balanced architectures. Consider, for instance, scientific computations. They involve matrix triangularization, matrix multiplication, grid computations of various dimensionalities, and also sparse matrix operations that have relatively high I/O requirements. Therefore in view of the results of Section 3, for scientific computations it is reasonable to assume the following:

$$M_{new} \geq \alpha^2 M_{old} . \tag{6}$$

That is, if the computation bandwidth of a PE is increased by a factor of α relative to its I/O bandwidth, then the size of the PE's local memory must be increased by a factor of at least α^2. For the rest of this section, we consider designing mesh-connected parallel computers for computations for which (6) holds.

On a parallel computer, a computation that is usually performed by one PE in a conventional serial machine is carried out by a collection of, say, p PEs. We can view this collection of p PEs as a new processing element that has p times as much computation bandwidth as the old PE. With this viewpoint, parallel processing is just a particular method of increasing the computation bandwidth of a PE. Therefore our methodology of re-balancing a PE by increasing its local memory applies directly to parallel architectures, as shown in the following subsections.

4.1. 1-Dimensional Processor Array

We want to use p linearly connected PEs to perform computations that were formerly done by a single PE, as depicted in Figure 3 below:

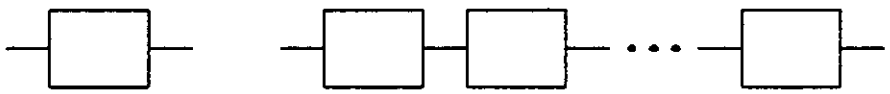

Figure 3. Using p PEs to perform computation formerly done by one PE.

The collection of p PEs can be viewed as a "new processing element" that has p times as much computation bandwidth as the original PE. The I/O bandwidth of

this "new processing element" is the same as that of the original PE, as only the two boundary PEs in the PE collection can communicate with the outside world. Therefore with respect to the "new processing element", the C/IO is increased by a factor of $\alpha = p$. This implies from (6) that the "new processing element" should have a total of at least p^2 times as much local memory as the original PE. That is, in the parallel arrangement, each PE should have at least p times as much local memory as the original PE. This translates to the following result:

> When using an array of linearly connected PEs for computations for which (6) holds, the size of each PE's local memory should grow at least linearly with the number of PEs in the array, to keep the array balanced.

4.2. 2-Dimensional Processor Array

We want to use $p \times p$ 2-dimensionally connected PEs to perform computations that were formerly done by a single PE, as illustrated in Figure 4 below:

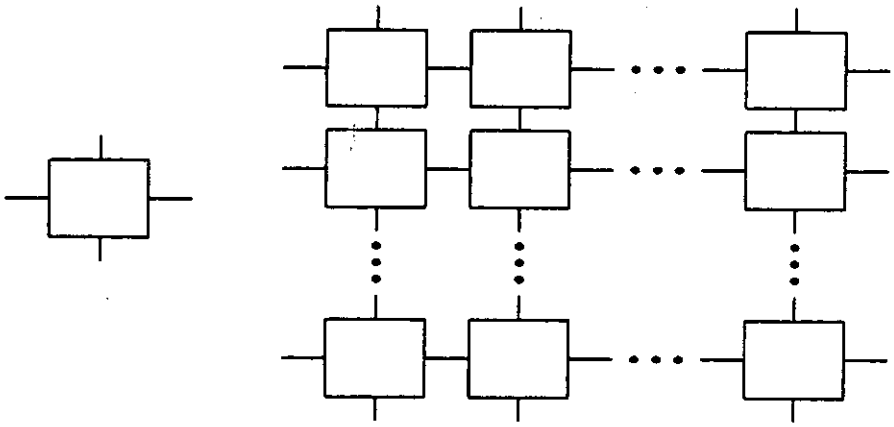

Figure 4. Using $p \times p$ PEs to perform computation formerly done by one PE.

By arguments similar to those used for the case of 1-dimensional processor array above, the computation and I/O bandwidths of this 2-dimensional array of PEs are p^2 and p times larger than those of the original PE, respectively. Therefore, C/IO is increased by a factor of $\alpha = p$. For computations such as matrix multiplication where (6) holds with equality, the parallel arrangement should have a total of p^2 times as much local memory as the original PE. This is automatically satisfied, since there are p^2 PEs in the parallel setup. Therefore, we have the following result:

When using a square array of mesh-connected PEs for computations for which (6) holds, it is possible to make the size of each PE's local memory to be independent of the number of PEs in the array, while keeping the array balanced. That is, the processor array is automatically balanced as more PEs with local memories of the same size are added to the array.

The possibility referred above depends on whether or not the computation can actually be decomposed for the parallel execution on the processor array. This is possible for example for matrix multiplication and triangularization, as demonstrated by various 2-dimensional systolic arrays for these computations [3, 7].

However, for computations (such as the d-dimensional grid computation with $d > 2$) where (6) holds with a strict inequality, an automatically re-balanced, square processor array is never possible. For these computations, the size of each PE's local memory must be increased as the size of the array increases.

5. CONCLUDING REMARKS

For most of the computations considered in this paper, to re-balance a PE, the size of its local memory must be increased much more rapidly than its computation bandwidth, *if the I/O bandwidth is kept constant.* For some computations such as the FFT and sorting, the local memory size must be increased exponentially as computation bandwidth increases. In this case, the size of the local memory may become unrealistically large, and the size of the application may also have to become unrealistically large in order to utilize all the memory. Therefore, for these computations one should not expect any substantial speedup without a significant increase in the PE's I/O bandwidth. Since increasing I/O bandwidth is difficult in practice, this partially explains why the performance of computer systems in general has not kept up with the rapid improvement in the computation bandwidth of processing elements.

For parallel architectures, we have shown configurations where each PE's memory should grow at least linearly with the number of PEs in the parallel system.

The CMU Warp machine [1, 4] consists of a 1-dimensional systolic array, which is an array of linearly connected, programmable PEs. With a local memory of up to 16K words, each PE can perform 10 million 32-bit floating-point operations per second, and transfer 20 million 32-bit words per second to and from its neighboring PEs. Having a rather large I/O bandwidth and a relatively large local memory for each PE of the Warp machine reflects the results of this paper.

The methodology and analysis techniques of this paper can be used for many other computations and architectures in addition to those considered here. Further work in characterizing other computations, in terms of their memory requirements for achieving balanced architectures, and in analyzing the impact of these results to various architectures, will certainly provide additional insights to the design of high-performance computers.

ACKNOWLEDGMENTS

Comments from Duane Adams, Allan Fisher, Monica Lam, Onat Menzilcioglu and Alan Sussman of CMU are appreciated.

REFERENCES

[1] Arnould, E., Kung, H.T., Menzilcioglu, O. and Sarocky, K.
A Systolic Array Computer.
In *Proceedings of 1985 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 232-235. March, 1985.

[2] Frandrianto, J. and Woo, B. Y.
VLSI Floating-Point Processors.
In *Proceedings of 7th Symposium on Computer Arithmetic*, pages 93-100. IEEE Computer Society, June, 1985.

[3] Gentleman, W.M. and Kung, H.T.
Matrix Triangularization by Systolic Arrays.
In *Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV*, pages 19-26. Society of Photo-Optical Instrumentation Engineers, August, 1981.

[4] Gross, T., Kung, H.T., Lam, M. and Webb, J.
Warp as a Machine for Low-level Vision.
In *Proceedings of 1985 IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[5] Gupta, A. and Toong, H. D.
An Architectural Comparison of 32-bit Microprocessors.
IEEE Micro 3(1):9-22, February, 1983.

[6] Hong, J.-W. and Kung, H.T.
I/O Complexity: The Red-Blue Pebble Game.
In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326-333. ACM SIGACT, May, 1981.

[7] Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.

[8] Siewiorek, D.P., Bell, C.G. and Newell, A.
Computer Structures: Principles and Examples.
McGraw Hill, New York, 1982.

[9] Song, S.W.
On a High-Performance VLSI Solution to Database Problems.
PhD thesis, Carnegie-Mellon University, Computer Science Department, July, 1981.
Also available as a CMU Computer Science Department technical report, August 1981.