# Memory Requirements for Balanced Computer Architectures

H. T. Kung

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

## Abstract

A processing element (PE) can be characterized by its computational 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 computation time equals the I/O time. Consider a balanced PE for some computation. Suppose that the computational bandwidth of the PE is increased by a factor of $\alpha$ 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 of avoiding 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 $\alpha^2$. For computations such as relaxation on a $d$-dimensional grid, the local memory must be enlarged by a factor of $\alpha^d$. For some other computations such as fast Fourier transform and sorting, the increase is exponential; i.e., the size of the new memory must be the size of the original memory to the $\alpha$-th power. All these results indicate that the size of a PE's local memory should be increased much more rapidly than the PE's computational 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.

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, in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007, and in part by a Shell Distinguished Chair in Computer Science.

## 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 computational 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 [3] and/or floating-point chips capable of delivering 10 million or more operations per second [7]. The computational bandwidth of such a processing element can be further increased by using multiple copies of these chips 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 I/O 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 computational 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 provide a mathematical framework to study the problem, the concept of balanced computer architectures is formally introduced in Section 2. Section 3 derives results on how much the local memory of a processing element must be increased as its computational bandwidth increases. Section 4 discusses implications of these results for some parallel computer architectures. Summary and concluding remarks are provided in Section 5.

## 2. Balanced Computer Architectures

As illustrated in Figure 2-1, we characterize a processing element (PE) by:

1. *C: the computational bandwidth*, which is the number of operations per second delivered by the PE;
2. *IO: the I/O bandwidth*, which is the number of words per second that the PE can communicate with the external

environment; and

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

This simple model is sufficient for the purpose of this paper. For an example of a more detailed model of computer systems, see [5, Section 2.5].

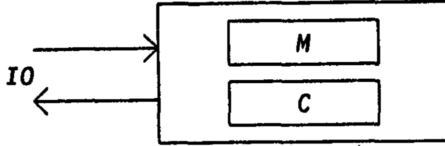


**Figure 2-1:** Processing element characterized by its computational bandwidth ($C$), I/O bandwidth ($IO$), and size of local memory ($M$).

In carrying out a computation such as fast Fourier transform (FFT) or matrix multiplication, a PE is said to be *balanced* if the I/O time equals the computation 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 computational bandwidth. Since it is usually difficult or expensive to increase the I/O bandwidth, we ask the following question:

> Consider a balanced PE for some given computation. Suppose now that the computational bandwidth of the PE is increased by a factor of $\alpha$ relative to its I/O bandwidth. That is, $C/IO$ is increased by a factor of $\alpha$. To rebalance 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") denote the total number of operations that the PE must perform for the computation, and $C_{io}$ ("cost for I/O") the total number of words that the PE must exchange with the outside world. Then the required computation and I/O times are $C_{comp}/C$ and $C_{io}/IO$, respectively. Therefore, the PE is balanced for the given computation if and only if

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

or

$$\frac{C}{IO} = \frac{C_{comp}}{C_{io}}. \quad (1)$$

Now suppose that $C/IO$ is increased by a factor of $\alpha$. Then by (1) *the PE is rebalanced if and only if the ratio $C_{comp}/C_{io}$ is increased by a factor of $\alpha$.* This provides a method of rebalancing 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 rebalance the PE. In the rest of the paper, we study by how much (expressed in terms of $\alpha$) $M_{new}$ 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 $\alpha$. This section derives 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 bound computations such as matrix-vector multiplication and solution of triangular linear systems: Impossible; i.e., PE cannot be rebalanced by merely enlarging its local memory, without increasing its I/O 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 $M$, the size of the PE's local memory.

### 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 uses no more than $M$ words of storage at any given time of the computation.

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[1], and $C_{io} = \Theta(N \cdot \sqrt{M})$ I/O operations.[2] Thus,

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

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

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

[2] Throughout the paper, we assume that one I/O operation transfers one word between the PE and the outside world.

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

The decomposition scheme we use here for matrix multiplication is just one of many possible ones. It has been shown [4] 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 the simpler problem of solving triangular linear systems. For example, this is the major step in all direct methods for solving linear systems. When $Q$ is restricted to be an orthogonal matrix, it is also the key step in computing least squares solutions and singular value decomposition, 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 annihilates those 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. 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,

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

It is easy to check that the same ratio is maintained for all the other 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 computing and image processing, this computation is usually called relaxation.

Assume that the computation is performed by an array of PEs. Each PE, with a local memory of size $M$, 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 FFT algorithm, assuming a local memory of size $M$.

Decomposition for FFT is not as straightforward as that for matrix multiplication and other computations considered above. Figure 3-1 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 that it can be entirely carried out inside a PE with $M$ words of local memory

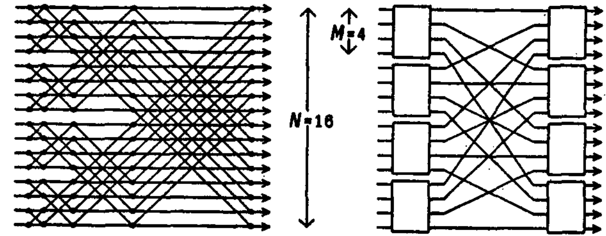


Figure 3-1: (a) 16-point FFT; (b) decomposition of the FFT computation.

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

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

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

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

It has been shown [4] that for FFT, any decomposition scheme yields:

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

where the function $l(M)$ 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 $\Theta(M)$, 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,

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

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 Bound Computations

All the computations considered so far have been computation bound, in the sense that computation takes more operations than I/O in order of magnitude. Computations that are not computation bound are called *I/O bound*. Matrix-vector multiplication and solution of triangular linear systems are examples of I/O bound computations. For I/O bound computations, after an increase of $C/IO$ for a PE, there is no way to rebalance 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. Classifying Computations by their Memory Requirements

The results summarized in the beginning of Section 3 suggests a classification of computations in terms of their memory requirements in achieving balanced architectures.

Consider, for instance, scientific computing. It typically involves 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, it is reasonable to classify scientific computing as a set of computations with the property:

$$M_{new} \geq \alpha^2 M_{old}. \quad (6)$$

Thus for scientific computing, if the computational bandwidth of a PE is increased by a factor of $\alpha$ relative to its I/O bandwidth, then the size of the PE's local memory must be increased by a factor of at least $\alpha^2$. When properties like (6) are explicitly stated for targeted computations, we will be able to evaluate architectures analytically as shown in the next section.

## 5. Implications for Some Parallel Computer Architectures

In 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 computational bandwidth as the old PE. With this viewpoint, parallel processing is just a particular method of increasing the computational bandwidth of a PE. Therefore our methodology of rebalancing a PE by increasing the size of its local memory applies directly to parallel architectures. This is shown in the following subsections.

### 5.1. 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 5-1.

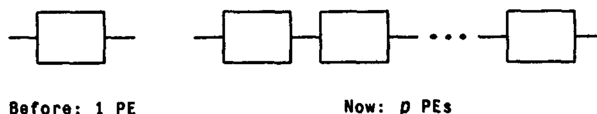


Figure 5-1: Using $p$ PEs to perform computation formerly done by one PE.

The collection of $p$ PEs can be viewed as a "new processing element" that has $p$ times as much computational 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 this "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.

### 5.2. 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 5-2.

We assume that only the $4p-4$ PEs on the boundary of the processor array can communicate with the outside world. By arguments similar to those used for the case of 1-dimensional processor array above, the computational 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 with equality, 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 to 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 [2, 6].

However, for computations (such as the $d$-dimensional grid computation with $d > 2$) where (6) holds with a strict inequality, an automatically rebalanced, 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.

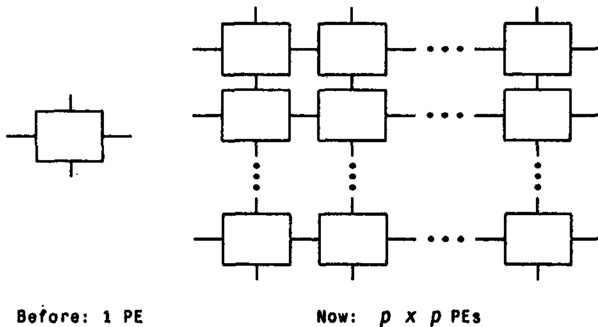


**Figure 5-2:** Using $p \times p$ PEs to perform computation formerly done by one PE.

### 5.3. Multi-dimensional Processor Array

Above results generalize in a straightforward way to $d$-dimensional processor arrays with $d > 2$. For example, one can show that for computations with the property that

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

a $d$-dimensional array of mesh-connected PEs is automatically balanced as more PEs with local memories of the same size are added to the array.

## 6. Summary and Concluding Remarks

For most of the computations considered in this paper, to rebalance a PE, the size of its local memory must be increased much more rapidly than its computational bandwidth, *if the I/O bandwidth is kept constant.* For some computations such as FFT and sorting, the local memory size must be increased exponentially as computational 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 take advantage of the large size of 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 computational bandwidth of processing elements.

In parallel architectures, a set of PEs are used to perform a given computation. For this set of PEs, the total amount of local memories and the aggregate I/O bandwidth bandwidth with the outside world should be balanced with the total computational bandwidth. We have shown configurations where the size of each PE's local memory should increase as the number of PEs devoted to a given computation increases.

The Carnegie Mellon Warp machine [1] consists of a 1-dimensional systolic array, which is an array of linearly connected, programmable PEs. With a local memory of up to 64K 32-bit words, each PE can perform 10 million 32-bit floating-point operations per second, and transfer 20 million words per second to and from its neighboring PEs. In particular the array can communicate with the outside world, via the PEs at the two ends, at a rate of 20 million words per second. 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 Carnegie Mellon are appreciated.

## References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986.

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

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

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

5. Kuck, D.J.. *The Structure of Computers and Computations*. John Wiley & Sons, New York, 1978.

6. Kung, H.T. and Leiserson, C.E. Systolic Arrays (for VLSI). Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

7. Nash, J.G. Review of Arithmetic Algorithms and Circuits for High Speed Digital Signal Processing. Proceedings of SPIE's O-E/LASE '86 Optoelectronics and Laser Applications in Science and Engineering, Society of Photo-Optical Instrumentation Engineers, January, 1986.

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*. Ph.D. Th., Carnegie-Mellon University Computer Science Department, July 1981. Also available as a CMU Computer Science Department technical report, August 1981.