Ann. Rev. Comput. Sci. 1986. 1:289–317

TYPE ARCHITECTURES, SHARED MEMORY, AND THE COROLLARY OF MODEST POTENTIAL

Lawrence Snyder

Department of Computer Science, University of Washington, Seattle, Washington 98195

> Likewise, when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.
>
> —General L. F. Manabrea (1842)

Manabrea's remark, referring to a design option for Babbage's Analytical Engine, has been cited (Hockney & Jesshope 1981) as the earliest reference to parallelism in computer design. The fact that Babbage considered parallelism allows us to conjecture that nearly a century and a half ago he understood the obvious, but nevertheless, Fundamental Law of Parallel Computation: A parallel solution utilizing p processors can improve the best sequential solution by at most a factor of p.

This law's truth follows from the observation that a speedup greater than a factor of p implies the existence of a better sequential solution. It provides an upper limit on achievable performance that has been difficult to approach in practice, much less to achieve. After only two decades of serious study (Hockney & Jesshope 1981) and only preliminary analysis of the limits to speedup (Hwang & Briggs 1984, p. 28), it is certainly premature to be pessimistic about our ultimate success at attaining the maximum predicted benefits of parallelism. Still, there are reasons to be cautious.

As a practical matter the scientific and commercial problems that are most in need of speedup are the so-called compute-bound problems (Bardon & Curtis 1983) since the I/O-bound problems would yield to better data transmission technology not more processing capability (Boral & DeWitt

8756-7016/86/1115-0289$02.00

1983). These compute-bound problems are generally superlinear, typically exhibiting sequential time complexity in the range $O(n^2)$ to $O(n^4)$ for problems of size n. The reason $O(n^4)$ problems are common in science and engineering is that they often model physical systems with three spatial dimensions developing in time.

A frequent challenge with these problems is not to solve a fixed-size instance of the problem faster, but rather to solve larger instances within a fixed time budget. The rationale is simple: The solutions are so computationally intensive that problem instances of an interesting size do not complete execution in a tolerable amount of time. Here "tolerable" means the maximum time the machine can be monopolized or the scientist can wait between experiments. Thus, with present performance the scientist solves a smaller-than-desirable problem to assure tolerable turnaround. The fact that the tolerable turnaround time will remain constant implies that increased performance will be used to attack larger problems.

What is the effect of parallelism on problem size? Keeping time fixed at t and (unrealistically) assuming the best possible speedup, it follows that superlinear problems can be improved only sublinearly by parallel computation. Specifically, if a problem takes

$$t = cn^x$$

sequential steps, and if the application of p processors permits an increase in the problem size by a factor of m,

$$t = c(mn)^x/p, \quad \text{then}$$

the problem size can increase by at most a factor of

$$m = p^{1/x}.$$

For example, to increase by two orders of magnitude the size of a problem whose sequential performance is given by

$$t = cn^4$$

requires, optimistically, 100,000,000 processors!

This observation, which I will call the Corollary of Modest Potential, is simply an interpretation of the Fundamental Law in the context of superlinear problems—the problems that matter. It states that in terms of extending our grasp, in permitting us to solve problems that are now too computationally intensive, parallel computation offers only a modest potential benefit. Notice, too, that the argument above is generous in focussing on low-degree polynomials. For interesting problems whose best sequential running time is of a higher degree or is exponential, the potential improvement is correspondingly more modest.

The Corollary was not formulated as an argument against parallelism. Parallel computation is perhaps the most promising way to improve computer performance now that technological advancements are approaching their inevitable physical limits. Rather, the Corollary has been formulated to emphasize that introducing overhead must be scrupulously avoided in the implementation of parallel systems, both in languages and in architectures. Because its benefit is so modest, the whole force of parallelism must be transferred to the problem, not converted to "heat" in implementational overhead.

This review assesses recent developments in parallel architecture and language in light of the Corollary's mandate for efficiency. Because the interface between these two elements is rarely smooth, the friction metaphor of ill-fitting parts producing heat generally applies. Thus, a secondary agenda item is to define a clean interface point between language and architecture from which both specialties may depart.

Preliminaries

Parallel computation means different things to different researchers, so it is important to delimit the range of interest here. A parallel computer is composed of multiple processor elements, each capable of executing a stored program, and used collectively to solve one problem. For present purposes their organization must generalize so the number of processors can grow without a serious performance penalty. (A small set of processors attached to a bus is not treated here because buses simply are not scalable.) This definition does not include distributed computer networks, pipelines or vector computers, or special purpose parallel computers embedded in larger systems.

To execute a parallel algorithm on a parallel computer requires that at least two encodings be performed:

$$algorithm \xrightarrow{p} program \xrightarrow{c} computer$$

where a programmer performs the first translation and a compiler performs the second. As we shall see, substantial overhead may be introduced with both encodings, but the importance of performance implied by the Corollary will motivate us to seek ways of avoiding it in both. In the final analysis, the parallel programming language—the medium in which the program is expressed, the target of the first translation and the source of the second translation—is the most critical element in effective parallel computation.

Language: The Medium is the Message

Although it is well known that languages like FORTRAN, LISP, APL and SNOBOL are universal in the sense of Turing computability, it is also acknowledged that each engenders a different programming style, each differs

in its suitability for programming a particular algorithm, and each favors particular problem solving methodologies. What makes these languages equivalent from a computability point of view and apparently dissimilar from the programmer's point of view is efficiency. The distinctions are annihilated in the grossly inefficient Gödel encoding that proves their equivalence, but the distinctions are exposed when writing an efficient program. The form of expression, the selection and use of control structures, and the type and management of data representations all influence the processing that can be efficiently and conveniently expressed. If sequential languages, or more generally sequential language constructs, influence the form and structure of the programs for sequential processors, it is reasonable to conjecture that parallel language constructs, the medium of expressing parallelism, control and shape the form and extent of the actions and interactions of parallel processors.[1]

Because this point—that language greatly influences the form and efficiency of a parallel program—is so central to the following discussion, it is necessary to be more specific: A programming language's semantics define an *abstract machine model,* or simply *model,* that is the conceptual device executing the language; the "instruction set" for this machine is the set of language constructs. So, ALGOL-like languages define a machine providing a nested naming environment with name scoping, recursion, call-by-name parameters, control structures, dynamically allocated arrays, etc (Naur 1963), while LISP-like languages define a machine providing a dynamic naming environment with S-expressions and lists, lambda binding, recursion, *map-car,* etc (McCarthy 1960). When the processing required for a particular algorithm matches that provided by a language's model, programming that algorithm will be convenient and the implementation will be efficient; matrix operations in ALGOL-like languages are a good example. But when the algorithm's requisite processing does not match that provided by the language's model, as with SNOBOL-like string matching expressed in APL or matrix operations expressed in pure LISP, the programming is difficult and the program is turgid and inefficient.

These difficulties are introduced by having to program around the language—that is, to implement constructs not in the language with the facilities of the language. The turgidity and inefficiency derive from two sources. First, the required instruction sequencing must be realized as a subsequence of those producible by the available control operators; and the required data repre-

[1]This last clause is a paraphrase of McLuhan's explanation of his famous dictum: " 'The medium is the message' because it is the medium that shapes and controls the scale and form of human association and action" (McLuhan 1964).

sentations must be realized by space-wasteful encodings using the available data structures, which do not capture the relationships needed for the algorithm. An extreme illustration is a matrix product program with the matrices implemented by S-expressions, *car*, and *cdr* rather than arrays with indexing, and the control implemented by recursion rather than *for* loops. Second, an extra level of interpretation is incurred when implementing a construct in any language as compared to having it available as a primitive.

Thus, the medium shapes and controls the actions of the machine because efficiency of execution and the convenience and clarity of programming select for those computations (a subspace of all computations) *directly* expressible with the language constructs.

Sequential Languages

It follows immediately, then, that a sequential language such as FORTRAN is an unlikely candidate for programming a parallel machine, not so much because FORTRAN favors a certain subclass of algorithms—every language will be restrictive in this sense—but rather because without parallel constructs in the language, the subclass is chosen from sequential algorithms. Expressing any computation in it entails specifying sequencing, whether or not it is essential to the formation of the result. Some of this sequentiality can be removed by systems such as Parafrase (Kuck et al 1980) and PFC (Allen et al 1983), thus permitting possible concurrent execution. The parallelism that is thus identified is constrained by the underlying sequential algorithm, and the chief difficulty remains: This mechanism (providing a sequencing and depending on the compiler to discover that the data dependencies do not require it) is the only means of specifying parallelism, and there are forms of concurrency that cannot be specified this way. The problem with sequential languages is they have no parallel constructs. (Notice that Parafrase and PFC were developed not so much to make FORTRAN a parallel programming language for new programs, as to extract whatever parallelism is available in extant FORTRAN programs.)

The implication that "parallel" algorithms are different from "sequential" algorithms warrants scrutiny. Since there seems to be no adequate formal definition, the intended distinction must be stated informally: Parallel algorithms exhibit the weakest possible ordering constraints, while sequential algorithms exhibit some artificial ordering restrictions. The distinction is nicely illustrated by the successive overrelaxation (SOR) computation used to solve linear systems of equations (Young 1971). A common implementation (Adams 1982) of this iterative technique using the "five point stencil" is to compute the $(k + 1)$st iteration from the kth iteration by traversing the SOR iteration array in row major order,

```
for i : = 1 to n do
  for j : = 1 to n do
    A[i, j] : = ω × A[i, j]
                + (1 − ω) × ((A[i, j − 1] + A[i − 1, j]
                + A[i, j + 1] + A[i + 1, j])/4);
```

The array is used simultaneously to store both old and new values. This provides some memory savings, but it serializes the row computations. That is, the program states that for a given sweep over the array, an element cannot be computed until the element to its left (and the one above it) have new values.

The data dependencies are such that the rows can be computed concurrently, provided they are offset by one position to the left in each successive row; the frontier between the $(k + 1)$st and the kth iteration is a 45° line "marching" across the array. Thus, an iteration takes $2n - 1$ steps.

Normally, the loop pair is followed by a test for convergence, but if this pair of loops is nested inside a *for* loop on k, the successive iterations can be done simultaneously, provided there are enough of them. That is, with full concurrency, $2n - 1$ steps are used for a start-up phase during which time successive 45° frontiers march across the array. This phase is followed by a steady state phase in which alternate array positions compute a new value on alternate steps. Finally, there is a $2n - 1$ step shutdown phase as the last iterations finish up (see Figure 1). This is probably the maximum amount of concurrency that can be achieved with the common SOR.

A more parallel alternative is known as the Red/Black SOR computation (Young 1950; Adams 1982; Adams & Jordan 1986). The name derives from visualizing the iteration array as a checkerboard and computing new values for all "red squares" on one step and all "black squares" on the next step (see Figure 2). Thus, the data dependencies permit a complete iteration to be computed in two steps.

Although the steady state of the fully concurrent common SOR appears similar in operation to the Red/Black SOR, these are very different algorithms. They have different data dependencies and hence cannot be derived from one another by reordering the evaluation. The particular order of evaluation of the common SOR leads to data dependencies that are not inherent in the solution of the problem. Thus, the common SOR has the characteristics of a sequential algorithm, while the Red/Black SOR, an almost perfectly concurrent algorithm, would be termed parallel.

Notice that this digression has sought only to illuminate the distinctions between sequential and parallel algorithms, and there is no claim that the Red/Black algorithm cannot be specified in a sequential language. In fact, Parafrase and PFC will recognize the FORTRAN text of Figure 3 as the

3	(3)	2	(2)	1
(3)	2	(2)	1	(1)
2	(2)	1	(1)	0
(2)	1	(1)	0	0
1	(1)	0	0	0

(a)

$k+2$	$(k+2)$	$k+1$	$(k+1)$	k
$(k+2)$	$k+1$	$(k+1)$	k	(k)
$k+1$	$(k+1)$	k	(k)	$k-1$
$(k+1)$	k	(k)	$k-1$	$(k-1)$
k	(k)	$k-1$	$(k-1)$	$k-2$

(b)

m	m	m	(m)	$m-1$
m	m	(m)	$m-1$	$(m-1)$
m	(m)	$m-1$	$(m-1)$	$m-2$
(m)	$m-1$	$(m-1)$	$m-2$	$(m-2)$
$m-1$	$(m-1)$	$m-2$	$(m-2)$	$m-3$

(c)

Figure 1 Common SOR, assuming full concurrency, showing the iteration number of values completed and (being computed); (*a*) in the start-up phase, (*b*) in the steady-state phase, (*c*) in the shutdown phase.

Red/Black data dependencies, although it is unclear whether this text is likely to appear in a program originally written for a sequential computer. The point of the digression is that the algorithms one formulates when thinking sequentially are qualitatively different from those formulated when exploiting parallelism.

One way to overcome the inadequacies of a sequential language is to add some parallel constructs to it. This common approach seems to have a potential hazard, assuming that the resulting language is actually an extension of the original rather than a whole new language with the old name. The approach requires that the extended language "retain" its sequential abstract machine model, since a program without parallel constructs must have the behavior of the original language's semantics. This can constrain the language design and influence the parallel facilities provided.

To illustrate the phenomenon, postulate an abstract machine model using a stack to maintain procedure activations. Retaining this stack-based model in the extended language implies that processes activated within a procedure must all terminate before the procedure returns. Thus, one cannot call a procedure to fire up a set of processes and then return to the main program. To have process activations with nonnested lifetimes requires a heap memory

(k+1)	k	(k+1)	k	(k+1)
k	(k+1)	k	(k+1)	k
(k+1)	k	(k+1)	k	(k+1)
k	(k+1)	k	(k+1)	k
(k+1)	k	(k+1)	k	(k+1)

Figure 2 Red/Black SOR showing iterations completed and (being computed).

organization for procedure activation and thus a fundamental change to the model. Of course the model can be changed provided it is equivalent to the stack when no parallelism is used. But if the model can be changed, why constrain the language design by the artificial requirement of preserving the semantics of a sequential language?

The argument is intended not as a proof that a sequential language is a poor starting point for parallel language design, but rather as an indication of the problem. "Add on" parallelism is not likely to be sufficient, and model development should not be constrained a priori to preserve sequential semantics. Just because a parallel language might reduce to a sequential language when there is only one processor, it does not follow that one can guess a good parallel generalization beginning with a sequential language.

```
      DO 10 I = 1, N-1, 2
        DO 20 J = 1, N-1, 2
20      A[I, J] = OMEGA*A[I,J] + (1 - OMEGA)*(A[I,J-1]
                + A[I-1,J] + A[I,J+1] + A[I+1,J])/4
        DO 30 J = 2, N, 2
30      A[I+1,J] = OMEGA*A[I+1,J] + (1 - OMEGA)
                 *(A[I+1,J-1] + A[I,J]
                 + A[I+1,J+1] + A[I+2,J])/4
10    CONTINUE
      DO 40 I = 1, N-1, 2
        DO 50 J = 2, N, 2
50      A[I,J] = OMEGA*A[I,J] + (1 - OMEGA)*(A[I,J-1]
               + A[I-1,J] + A[I,J+1] + A[I+1,J])/4
        DO 60 J = 1, N-1, 2
60      A[I+1,J] = OMEGA*A[I+1,J] + (1 - OMEGA)
                 *(A[I+1,J-1] + A[I,J]
                 + A[I+1,J+1] + A[I+2,J])/4
40    CONTINUE
```

Figure 3 Serial program text defining the Red/Black SOR Data Dependencies.

Parallel Languages

The goal in parallel programming language design, then, is to discover a conceptually clean abstract machine model that matches the form of a coherent subclass of parallel algorithms in the same way FORTRAN, LISP, APL, SNOBOL, etc match the algorithms they support well. But one other major constraint governs parallel language design besides the closeness of the match to a parallel algorithm subclass: the efficiency of the program's implementation on a specific machine.

The problem is that different sequential languages can be implemented with acceptable efficiency on any sequential machine; but there is a much greater diversity among parallel machines, making the matter of efficient implementation of a particular language on a particular machine problematical. Consider the options.

A high level, abstract language that does not presume particular features of the underlying machine has the advantage of being portable between machines. It has the disadvantage of not being able to exploit particular hardware features, except insofar as the compiler is able to recognize the opportunity to do so; furthermore, the implementation of the high-level abstractions is likely to incur considerable overhead. A low-level language with machine-specific constructions will presumably exhibit good performance and low overhead on some machines; but if it is worth porting to other machines at all, it is not likely to run well because of the software implementation of the features available in hardware on the preferred machines. The desire for high level abstract languages and the need to be efficient pose an apparently insurmountable gap to be covered by compiler technology.

Type Architecture

At the heart of the problem is the fact that there is no widely accepted parallel analogue to the sequential von Neumann machine. There is no idealized set of facilities that all physical machines tend to provide more or less efficiently and that compiler writers can expect to find in their target computers. If there were such an idealized parallel machine, then the dissimilarities among parallel computers just mentioned might be neutralized by architects who would make the diverse physical structures realize the idealization. We refer to such an idealization as a *type architecture*[2] and note that it is a standardization of the hardware/language interface.

A type architecture is not a rigid specification to which all architectures must conform and all languages adhere. It defines a region of consensus,

[2]The term is analogous to *type species* in taxonomy, the species of a genus with which the generic name is permanently associated.

being explicit about a few salient features and mute on everything else. For example, the von Neumann machine treats the concepts of a stored program, a program counter, the memory-processor relationship, etc and ignores addressing modes, general purpose registers, virtual memory, etc. A type architecture, by establishing agreement on the main points of interest, reduces the differences to mere details. Notice that we have no reason to expect a single parallel type architecture to suffice. On the other hand, a proliferation of type architectures would not be desirable for two reasons. First, it would probably signal a failure to find the correct, unifying concepts; and second, if there actually are many fundamentally different ways to compute in parallel, then wide program portability may be hopeless, since it is doubtful that programs written in a language based on one type architecture would be convertible to efficient programs for machines based on another.

Although no parallel type architecture has achieved wide acceptance, the paracomputer (Schwartz 1980) has developed a substantial following. It has been embraced by the theory community, where it is called the PRAM; more importantly, it has been used implicitly as a type architecture by advocates of extended sequential, high-level, and machine-independent parallel languages. However, it is an inappropriate choice for general parallel computation, as Schwartz apparently recognized and as I will demonstrate below, because it is unrealizable. Thus, in terms of defining a reference point between the abstractions of language and the possible realizations of architecture, it is too abstract, too far from what can actually be built.

Evaluating the Paracomputer

The case against the paracomputer as a type architecture begins with a concise example of its failure to serve that role well. Recall that the paracomputer is an idealized, shared-memory multiprocessor:

> One such model is that in which a very large number p of identical processors (each with a conventional order-code) share a common memory which they can read simultaneously in a single cycle. In such a model we also assume that during any access cycle any number of processors can simultaneously write to memory, and that a memory cell to which writes are simultaneously addressed by many processors will come to contain one of the quantities written to it (perhaps a randomly selected one of these quantities, or perhaps the minimum). We call this very general model of parallel computation the *paracomputer* model (Schwartz 1980).

Postulate a programming language based on the paracomputer, and suppose that the language is transparent in the sense that it has language constructs that permit a collection of processors to access the flat, shared memory with unit access time. A programmer, wishing to write a program to find the maximum of n elements with $n = p$ processors, will discover in a search of the literature (Shiloach & Vishkin 1981) that Valiant's algorithm (Valiant 1975) is optimal

for this set of assumptions. The running time for this algorithm on the paracomputer is O(loglog n), as will be seen, but the performance of the program on a real machine will not be so good.

The algorithm operates in stages; at the s stage, the $n(s)$ distinct input values $a_1, \ldots a_{n(s)}$ are partitioned into the fewest number r of sets $S_1, S_2, \ldots S_r$, of essentially equal size ($abs(|S_i|-|S_j|) \leq 1$), such that

$$\sum_{i=1}^{r} \binom{|S_i|}{2} \leq p \ .$$

For each set S_i, $\binom{|S_i|}{2}$ processors are assigned so that each processor compares one of the $\binom{|S_i|}{2}$ distinct pairs of elements. The result of the comparison is recorded in an auxiliary Boolean array $b_1, \ldots, b_{|Si|}$ which is initialized to all 1s at the start of the stage: Processor p assigns 0 to b_k where k is the index of the smaller element of the pair. This operation requires concurrent writes to memory locations. It takes one step to find the maximum of the set, and the winner is identified as the (sole) element a_l with a corresponding $b_l = 1$; that is, it was not the smaller of any of the $\binom{|S_i|}{2}$ distinct pairs of the set S_i. Each winner of a set at stage s becomes an input to stage $s + 1$. The maximum of any set of n values can be found in loglog $n + c$ stages. The bookkeeping details of the algorithm, which can be performed in constant time for each stage, are described in Shiloach & Vishkin (1981).

As an example, the maximum of a set S of 1000 elements can be found as follows:

stage 1: S is divided into 332 sets of 3 elements each and 2 sets of 2 elements each; each 3-set requires $\binom{3}{2} = 3$ processors and each 2-set requires $\binom{2}{2} = 1$ processor; a total of $998 = 996 + 2$ processors are used to produce a set S' of 334 winners.

stage 2: S' is divided into 46 sets of 7 elements each and 2 sets of 6 elements each; each 7-set requires $\binom{7}{2} = 21$ processors and each 6-set requires $\binom{6}{2} = 15$ processors; a total of $996 = 966 + 30$ processors are used to produce a set of S'' of 48 winners.

stage 3: S'' is divided into 2 sets of 24 elements each which require $\binom{24}{2} = 276$ processors each to find the maximum; a total of 552 processors are used producing 2 winners.

The maximum of the two values is computed directly and returned without another stage. Notice that the comparison depth of 4 is superior to the comparison depth of 10 for the obvious binary tree algorithm.

As I'll explain below, the memory accesses that are assumed to take unit time for the paracomputer cannot be performed that rapidly on any physical machine. In fact, they will take at least $\Omega(\log n)$ time when the program runs—perhaps more—depending on the machine. Thus the observed running time of the program will be at least O(log n loglog n) on any physical

machine. Contrast this with the fact that the same physical machine could realize O(log n) performance on this problem: Assuming that the architecture "contains" a binary tree, that is, the processing elements can directly communicate in a manner described by a binary tree, then they can percolate the maximum to the root in O(log n) time.

This is a serious matter. A machine that can find the maximum in O(log n) time requires O(log n loglog n) to run the algorithm that is optimal for the paracomputer. The language, and by extension the type architecture on which it is based, have been harmful rather than helpful. Indeed, this maximum-finding example seems to be an instance of the earlier claim that the language influences the choice of algorithms, and here the language led the programmer to the wrong choice.

Unrealizability of the Unit Cost

At the heart of the maximum-finding example is the claim that the facilities used in the optimal algorithm cannot be implemented with a constant unit cost. Moreover, it was claimed that these operations actually require at least $\Omega(\log n)$ time on any physical realization. Both of these claims are widely accepted, but it is still worthwhile to examine them closely.

The unit-cost assumption means that independent of the number of processors and independent of the characteristics of the reference sequences performed by the other processors, there is a constant upper bound on the time for any processor to reference any memory location. This condition is obviously unrealizable in practice because among other things it violates the physical law that an arbitrary amount of information cannot be stored within a fixed transmission time of a point. But this is not a compelling justification for claiming the unrealizability of the unit-cost assumption, because similar reasoning would also compromise the widely accepted unit-cost memory-reference assumption for von Neumann machines with arbitrarily large memory. Since it is more beneficial than harmful to ignore this physical limitation for the von Neumann type architecture, it seems constructive to do the same here.

To translate this simplification into a form that is most useful to the present discussion, let us assume unit-time transmission over an arbitrarily long data path. This permits the components of the parallel computer to be physically separated, as they must be, without that requirement's influencing the cost analysis out of all proportion to its actual importance. For distances encountered in a parallel computer, the contribution of the physical distance to the transmission delay is so small at the speed of light as to be negligible.[3]

[3]There are settings in which the transmission delays are substantially larger than those predicted by speed of light estimates. MOS VLSI chips are one example. But because other technologies (for example, ECL) do not have these problems, it seems justified that the type architecture be independent of this technological consideration.

Though the relative importance of distance can change with technology, for now we will assume information transfer is not tied directly to distance.

The unrealizability has more to do with the multiplicity of processing sites than with distance: Each of p processors is physically separated, each of the m memory locations is also physically separated, and the model of computation requires information to move between arbitrary pairs of these sites at each time step. To access an arbitrary memory location in unit time requires that a processor not collide with any other access, lest the collision produce a delay that causes the time unit to be exceeded.[4] Thus each processor must have an independent path to each memory cell. Furthermore, there must be no decision logic along the path, lest switching delays cause the time unit to be exceeded. Thus, each processor must have a direct path to each memory cell. Consequently, the unit-time property implies the need for mp direct, independent paths.

There are two essentially different approaches to implementing these independent paths: Separate physical paths, m incident with each processor and p incident with each memory cell, violate the physical constraint that an arbitrary number of paths cannot fan into or out of a point. A single physical path connecting the processors and memories transmitting p data values in unit time violates the physical constraint that an arbitrary number of signals cannot be simultaneously multiplexed on a single datapath. A strategy based on a bounded degree or a bounded amount of multiplexing or a mixture of both leads only to a bounded number of direct, independent paths. Thus, the unit-time access to memory cannot be achieved.

Though the paracomputer's unit-cost shared-memory property is generally unrealizable, physical machines have been built around these two techniques. The separate-physical-paths solution or cross bar is illustrated by the C.mmp (Wulf & Bell 1972) architecture where $p = 16$. The other extreme, the single physical path solution or bus, is illustrated by the Sequent (Fielland & Rogers 1984) architecture, where $p = 12$. It is unclear how large p can be practically, but since cacheing is likely to be a critical component, recent analyses by Archibald & Baer (1985) suggest $p = 64$ is a serious barrier. Although the unit-time constraint cannot be realized for arbitrary machines, the memory sharing can be. With bounded fan-in and fan-out, a tree must be used to raise the degree of both the processors and memories, engendering an $\Omega(\log m)$ access delay. This justifies the logarithmic performance degradation mentioned in the maximum-finding example.

[4]It is understood throughout this discussion that prohibitions such as "processor not collide" should properly be expressed as "processor not collide more than a bounded number of times" in order to provide for the possibility of violating the prohibition by a small but limited amount which can be absorbed into the time unit. Because it increases clarity without losing generality, we give the less tedious but more rigid condition.

Notice that in this analysis m refers to individual memory locations. It is customary, of course, to group many cells together into a memory module and treat them as a single unit. This reduces the value of m, yielding generally favorable architectural consequences (reduced hardware) and generally unfavorable performance consequences. First, because a memory module cannot be used simultaneously by more than one processor, there is a delay for all but one of the processors, whenever two or more accesses to a module occur at the same time. Second, there is overhead in arbitrating the possible multiple requests. Third, decoding the address, to find the proper memory location, requires $\Omega(\log n)$ time for an n word memory module. Since the constant is small, it can with little risk be ignored as with the von Neumann type architecture. Notice that all three of these performance-degrading properties increase with n, and the first is quite serious.

Type Architecture's Influence on Language

With its unrealizability established, the role of the type architecture in leading to the wrong maximum-finding algorithm can now be analyzed. Recognize that what is crucial here is the type architecture's role in defining a region of consensus between languages and machines; the issue is not simply that the paracomputer cannot be built. Indeed, its unrealizability has been widely acknowledged, but many scientists have gone on to argue that the paracomputer is just the kind of high-level language abstraction that contributes to effective and efficient programming. It should therefore be used even if the implementation gap must be covered by software. Obviously it doesn't contribute to efficient and effective programming. The purpose of this and the next section is to explain why.

The fallacy in this argument—that the paracomputer is an ideal, high-level abstraction promoting effective programming—is that the paracomputer is not used here or elsewhere as an abstraction, but rather as a type architecture. The distinction is important: A language abstraction is an idealized structure expressed in terms of other basis facilities. A type architecture *is* the basis facility.

The type architecture is the machine on which a language's abstract machine model is implemented. That is, in formulating a language's semantics and in expressing how the compiler is to implement the semantics, the role of the type architecture is that of target machine. This is completely appropriate because on the hardware side, physical machines will implement the type architecture's facilities. Thus the target machine on which the language model "runs" will exist.

The language's abstract model might be quite transparent, providing facilities that correspond directly to type architecture facilities. This is not a level of interpretation, but rather is a means by which source language features can

correspond one-to-one to machine language features; for example, *gotos* correspond to unconditional jumps. Alternatively, the language could provide one or more layers of abstraction implemented on top of the type architecture. The cost of this implementation, expressed in terms of type architecture facilities, is a fair statement of the cost of using the abstraction in a program, since architecture of this type can be directly implemented in hardware. So, the type architecture permeates the language, either explicitly or implicitly in the implementation of the language constructs.

Paracomputer's Influence on Language

The shortcomings of the paracomputer as a type architecture can now be explained in terms of the two translations mentioned earlier, the programmer's translation p from algorithm to program and the compiler's translation c from program to linked object code.

THE PROGRAMMER'S TRANSLATION In the maximum-finding problem, the programmer selected the algorithm that was optimal with respect to the unit-cost assumption of the language. This assumption in turn was simply inherited from the type architecture because the language was assumed to be transparent. Given that the problem of finding the maximum can be solved in $O(\log n)$ time on a physical machine, and (optimistically) accepting that the program can be translated to run on a physical machine with $O(\log n \log\log n)$ execution time, the algorithm was not optimal for the actual existing conditions, and the programmer was poorly served by the type architecture. At a minimum, the programmer should have been told that the cost of each step was at least $\Omega(\log n)$. This means that every program would have had this factor in its running time, so that to achieve an $O(\log n)$ result, the original time complexity would have had to be constant. There is no constant-time maximum-finding algorithm for the shared memory unit-cost model. [Recall that for the $n = p$ assumption the $O(\log\log n)$ algorithm is best.] *It is impossible, therefore, to write a maximum-finding program in any language based on the paracomputer type architecture and achieve the best physically achievable execution time.* This leads to a fundamental conclusion:

- A type architecture must accurately reflect costs.

When it does not, the cost of the language's facilities will be biased, and the programmer will be unable to assess the running time of the program. Algorithms thought to be optimal will not be. Worse yet, certain performance cannot be achieved, no matter how much ingenuity is applied.

To emphasize, the key point is not the magnitude of the difference but rather the provable necessity of a gap between the best physically realizable performance and the best physically realizable performance of a program

written in a paracomputer type language. Indeed, had care been taken to charge for the facilities actually provided by the paracomputer model, such as concurrent reading and writing, the gap would probably have been larger by at least a factor of O(log n) (Snir 1985; Cook et al 1986). (Answering the question exactly takes us into the pointless activity of physical analysis of formal models of computation.) Also, the fact that the paracomputer has concurrent read and write is relevant only to the details of this particular presentation, and not relevant to the fundamental argument; any model misrepresenting important costs will suffice.

THE COMPILER'S TRANSLATION Postulate a programming language based on the paracomputer as a type architecture, and consider the problem of compiling to some physical machine. The problem is substantial because the language designer, who regarded his job as complete when he expressed the language's abstract model in terms of the paracomputer, left a large gap between the paracomputer facilities and the physical machine. (The gap doesn't exist for a true type architecture, of course, since it can be implemented.) Moreover, the gap must be "spanned" differently for each physical architecture unless someone takes the time to express the paracomputer's semantics in terms of a small set of facilities generally available on physical parallel computers (in which case this small set of facilities would be called a *type architecture*, the paracomputer would be called an *abstraction*, and the objectives we are advocating would be achieved).

To appreciate the problems caused by the paracomputer from the compiler's point of view, assume the target machine is a nonshared-memory parallel architecture. This may at first seem foolish, but unlike architectures with interconnection networks having all processors equidistant from each other (see next section), most nonshared-memory architectures have processors separated by varying distances. The ultracomputer[5] for example, has all processors connected in a shuffle graph and thus the maximum distance between two processors is log n; the direct communication is but one. Most importantly for the present discussion, this minimum distance can be exploited so that most nonshared memory machines can find the maximum of n values in O(log n) time. [In fact, they can find the maximum of n log n elements in O(log n) time (Schwartz 1980)!]

Suppose that the programmer writes the appropriate binary tree percolate algorithm. If the compiler implements the literal shared-memory facilities,

[5]Notice that the ultracomputer (Schwartz 1980) and the NYU Ultracomputer (Gottlieb et al 1983) are radically different machines: The former is a shuffle-connected nonshared-memory machine, the latter uses an omega network to connect processors to shared memory modules. The reasons for the difference seem to be largely historical (Gottlieb 1981). The reader should note the distinction and be advised that both machines are mentioned in these pages.

this algorithm will take at least $\Omega(\log^2 n)$ time. Instead, the hope is for the compiler to implement the program in the ideal way, taking advantage of features of the underlying machine. To do so, the compiler must know that the values should be mapped onto the processors, one value per processor, and that the processing that percolates values towards the root should be allocated so that nodes adjacent in the tree are adjacent on the underlying architecture. It is generally impossible for the compiler to deduce such information from a program, of course. If there is any hope of having the compiler achieve the $O(\log n)$ efficiency, then, the programmer must tell the compiler how to allocate memory, how to allocate processing, how to schedule I/O, etc. But the paracomputer has undifferentiated processors and undifferentiated memory. Structural components of the underlying machine are not visible from the program. There is no way to correlate entities of the source program with them.

Using a "blind" scheme where the programmer specifies generic allocation and assignment information will not work for two reasons. First, without knowing the underlying machine, one doesn't know what to specify to aid the compiler. Second, without knowing the underlying machine, one cannot judge whether the postulated allocations are feasible, and their feasibility affects the choice of problem-solving techniques. This leads to a second fundamental conclusion:

- The type architecture must display the principal structural features of the architectures.

Features hidden by the type architecture are hidden from the language; therefore the language designer cannot incorporate facilities for describing their use; without such facilities the programmer cannot tell the compiler how the program should be run, and with no help from the programmer the compiler must employ only generalized, inefficient translations.

The conclusion is that the paracomputer cannot be used as a type architecture because it fails to reflect accurately the costs of physical machines and it fails to describe important structural features of physical machines. A candidate type architecture, to serve in the paracomputer's stead, is described in a later section. Though it may have difficulties, the candidate type architecture meets these two requirements.

As a postscript to our criticism of the paracomputer, some loose ends should be tied up. First, our criticism of Schwartz's paracomputer has been directed at the way it, or an equivalent machine, has been used by others; Schwartz (1980) used it as a theoretical tool and employed the ultracomputer in the role of what we would now describe as a type architecture. Second, the inadequacies of the paracomputer as a type architecture do no more to diminish the role of its theoretical equivalent, the PRAM, than the unrealizability of the Turing Machine does to diminish its importance as a

theoretical tool; this value lies in what they tell us about the nature of computation. Valiant's algorithm illustrates this point nicely. It seems to imply that the complication [O(log n) time] in finding the maximum is more attributable to the time required to bring the values together [Ω(log n)] than to the actual accumulation of the information since, when the time to bring the values together is removed by the paracomputer model, the time to accumulate the information turns out to be smaller [O(loglog n)].

Shared-Memory Computers

Although we have already noted that the direct implementations of the shared-memory facility have a serious limitation as the number of processors approaches 64, the indirect implementations have not yet been considered. These are the so-called dance hall architectures, a name assigned by Keller (1982) alluding to their characteristic structure of a set of processors lined up on one side of a processor interconnection network (dance hall) and a set of memory modules lined up on the other side. Recently proposed dance hall architectures include the NYU Ultracomputer (Gottlieb et al 1983), the PASM Computer (Siegel et al 1981), the Cedar Computer (Gajski et al 1983), the Butterfly (Crowther et al 1985), and the RP3 (Pfister et al 1985). The salient property, it seems to me, is that all processors are "equidistant" from each other through the shared memory, and so cases like the Butterfly, where the memory wraps around to the processors giving each a direct connection to one module, are still dance hall architectures. (The Cedar machine has a second, direct path for processors within a cluster, but is still a dance hall architecture.)

The interconnection networks used for dance hall architectures typically have log p depth (Siegel 1985) and thus each reference to shared memory has at least Ω(log p) delay. This delay is not like the trivial address decoding overhead which we argued earlier could be ignored; it is significant (Franklin & Dhar 1986) in current technologies even though every effort is made to make it as small as possible (Gajski et al 1983).

The corollary of our earlier argument is that dance hall architectures find the maximum of n values in O($\log^2 n$) time; the reader should be cautioned, however, that although this conclusion correctly implies superiority of non-shared-memory architectures, this is to date only theoretically true. No non-shared memory machine has been engineered to have processor-to-processor communication execute as fast as the typical shared memory access of the dance hall architectures.

A more serious problem with the interconnection network structure is that the Ω(log p) performance is the collision-free performance and can be much worse when collisions arise. Borodin & Hopcroft (1982) have shown that a

deterministic (oblivious) routing strategy can take $\Omega(\sqrt{p})$ time in worst case to route messages across an interconnection network because of collisions.

Many strategies have been developed to try to reduce collisions or their effect. The combining switch of the NYU Ultracomputer (Gottlieb et al 1983) provides the ability to merge colliding requests to the same memory location. Clever allocation of data structures can avoid collisions when standard reference sequences are used (Lawrie 1975). Architectural techniques can hide memory latency (Smith 1981). General mechanisms have also been studied to control the hazards of collisions (Valiant & Brebner 1981; Upfal & Wigderson 1984).

Alternative Type Architecture

Criticizing the paracomputer as a type architecture is considerably easier than offering an alternative. Still, it is instructive to present a candidate parallel type architecture in order to illustrate a contrasting set of issues. Consider a Candidate Type Architecture:

> *CTA: A finite set of sequential computers connected in a fixed, bounded degree graph, with a global controller.*

Although this architecture has been widely used in the technical literature for presenting nonshared-memory parallel algorithms, its characteristics must still be amplified.

> *finite set:* This limitation, though necessary in practice, is analogous to the memory-size limitation in von Neumann machines—that is, it is no limitation at all. Programmers assume there is no bound and the compiler handles the case when it is exceeded (see section on exploiting graph properties).
>
> *sequential computers:* A von Neumann type architecture is presumed, and hence a locally stored program and data; the computers run asynchronously. The term "sequential" means "implements a sequential instruction stream" and is not intended to exclude parallelism in the form of coprocessors, pipelining, or other limited parallelism such as instruction packing (Fisher 1983) or multi gauging (Snyder 1985).
>
> *connected:* The term is intended to connote the ability to send a simple value between computers efficiently (i.e. in small constant time) and the ability for each computer to transmit to its adjacent neighbors simultaneously.
>
> *fixed, bounded degree graph:* The sequential computers are the vertices of the graph and the connections form the edges of the graph; the fact that it is a graph and not a hypergraph excludes buses; the bounded degree limitation is self-evident; the fixedness of the graph acknowledges the fact that a machine must be wired together in some permanent form.
>
> *global controller.* This is a sequential machine that can broadcast signals (for example, reset) or perhaps single values to all the computers, can address individual computers to

request single values, and can interrupt or be interrupted by them. This facility provides a weak control over the computers consistent with the multiple independent programs and asynchronous control.

The nature of the type architecture is such that many aspects of parallel computers are omitted in the CTA definition. For example, whether the processors operate synchronously or asynchronously is not mentioned, but since asynchronous is more general, it must be assumed. I hope the features that have been defined are the critical ones for establishing a working consensus between architects and language designers.

The CTA defines, by inheritance from the von Neumann type architecture, unit-cost memory access to a local memory; it is mute on the existence of any global memory. The processors are independent, communicating with a bounded number of neighbors over channels represented by the edges of the graph; they are capable of fine-grain communication, since transmission of simple values is explicitly mentioned. The small constant time assessed for interprocessor communication, though technically not achievable in the limit, is consistent with the earlier argument that data transmission is best viewed as independent of distance. Evidently, the CTA overcomes the problems of the paracomputer, since it assesses costs fairly and exposes the underlying structure.

CTA Architectures and Languages

Recall that a type architecture is not a rigid specification to which all architectures must conform and all languages adhere; rather, it defines a region of consensus. Thus architectures and languages may be evaluated in terms of how closely they match the consensus. In the case of architectures, this means evaluating how efficiently they provide the facilities mandated by the type architecture. In the case of languages, it means assessing whether the abstract machine model would run on the type architecture. The following (nonexhaustive) review should solidify the intended meaning of the CTA.

The ultracomputer of Schwartz (1980) is obviously an instance of the CTA, though the global controller is not explicitly mentioned. The Cosmic Cube (Seitz 1985) efficiently provides the requisite facilities of the CTA. It is not an *instance* of the CTA, because the binary n-cube architectures have vertex degree (that is, the number of processors to which each processor is connected) of $\log n$, and the CTA specifies bounded degree independent of the number of processors.[6] The CHiP architecture (Snyder 1982) is another example of a generalized CTA; its configurability permits the CHiP computer to implement a variety of graph interconnections.

[6]Pursuing the taxonomic analogy further, the Cosmic Cube is in the CTA genus, but it is not the species that gives the genus its name.

It is permissible for machines to differ from (in this case generalize) the type architecture, as long as they provide the type-architecture facilities efficiently. The opposite is true for languages; a language can be based on the type-architecture facilities, or a subset of them, but it is not a type language if it is based on a super set. Thus,

$$\textit{facilities}(\text{type architecture}) \subseteq \textit{facilities}(\text{physical machine})$$

$$\textit{facilities}(\text{type language}) \subseteq \textit{facilities}(\text{type architecture})$$

describe the constraints imposed by a type architecture.

Systolic arrays (Kung & Leiserson 1980) are not architectures of the CTA type, since the processors are not von Neumann machines and the array is synchronous. Physical implementations have tended to use von Neumann machines as processing elements but retain the synchronous property (Dohi et al 1985; Bromley et al 1981). Some machines are excluded from the CTA genus by virtue of being single-instruction stream computers (Bouknight et al 1972). Of course, the dance hall architectures are excluded because the processors are unconnected, communication must go through the log n depth network to the shared memory, and thus they fail to implement the bounded time communication requirement. The important property of all these architectures is not that they are different from the CTA but that they do not provide the facilities of the CTA with the requisite efficiency.

These remarks have applied to architectures and are thus quantified over a family of machines. One effect of such quantification is that it confuses the properties of a whole family with the properties of individuals. Thus every single instruction stream, multiple data stream (SIMD) architecture is unable to run programs from CTA-type languages because these programs generally require multiple instruction stream; each machine fails to be of the CTA type. By contrast, dance hall architectures can run programs from CTA-type languages quite efficiently until the number of processors gets so large that the logarithmic depth of the interconnection network cannot be treated as a constant; the individual machines are acceptable but they do not scale.

To evaluate languages in terms of the CTA, one must determine whether the language's model can be implemented on the CTA. This determination is complicated by the fact that the CTA is universal in the sense of Turing, so it can host any programming language, though to do so may require running the program on one processor. To avoid this problem and to bring the discussion into the realm of practical programming languages, agree that the test of whether a language is of CTA type is whether the language's model has been implemented on a CTA-type architecture, or whether it is known how to do so.

Cosmic Cube C (Su et al 1985), the language designed for the Cosmic

Cube, is a CTA-based language if, as it appears, it does not rely significantly on the non-CTA property of the Cosmic Cube, the unbounded degree. Poker, (Snyder 1984a), the language designed for the CHiP architecture, is also a CTA-based language. OCCAM (INMOS 1984) is in principle a CTA-type language.

An interesting kind of language qualifying as a CTA type is one developed for a more limited class of architectures; programs written in such a language must, by containment, run on a CTA. One example is Crystal (Chen 1986). This system was developed to generate systolic array programs from a set of recurrence equations. Since systolic arrays can be well approximated by the CTA architecture—all processors running the same code is equivalent to the SIMD operation, and regular (coarse-grain) signals from the controller can keep all the asynchronous processors adequately synchronized—the programs should run on a CTA.

Most well-known parallel programming languages are not CTA-type languages, since it is apparently not known how to implement them efficiently on a CTA-type machine. These include the data flow languages [e.g. VAL (Ackerman 1982)], which treat variables in a flat, undifferentiated name space. Extended sequential languages, for example FORTRAN 8X, are also not CTA-type languages since they preserve the shared memory of their namesake. Also, high-level parallel languages, for example Concurrent Prolog (Shapiro 1984), have the added difficulty of a complex language model. All of these languages can possibly be converted into CTA-type languages by implementing their language models on the CTA.

Evaluating the CTA

Although it meets the requirements of being cost accurate and structurally explicit, there are other considerations, and so the CTA is offered only as a candidate type architecture. Consequently, it is appropriate to evaluate the CTA to see what it does and does not offer.

One apparent shortcoming of the CTA is the absence of any specific choice of graph. The justification for this decision is that no graph has emerged as the ideal, and to choose one now would simply be premature. Among the choices that could have been made are the shuffle exchange graph as used in the ultracomputer; the Cube Connected Cycles graph of Preparata & Vuillemin (1981), and other "universal" graphs (Siegel 1985); the array-type structures, including the 8-, 6-, and 4-connected meshes and toruses and the linear (2-connected) arrays; and finally trees of various species. Each graph family has its assets and liabilities, and so far we've had too little experience to make an optimum selection.

The unspecified graph structure is probably a benefit from the architect's point of view since it places no a priori constraints on design decisions.

Language designers, on the other hand, do not appear to be well served by this feature of the CTA. This may be more an appearance than a reality, though, because designers must inevitably begin dealing with a graph-based type architecture, a more difficult medium than the malleable paracomputer. Having a specific graph to work with hardly seems a great simplification.

The language designer's task is clear: Either implement an existing language's abstract model on the CTA or develop a new language and implement its model on the CTA. The implementation of an existing language might be favored because of a large body of existing software, but consider what must be done: The designer must define how variables and data structures are to be allocated to the local memories of the specific machines by a compiler. In addition, the revised language model must explain how references to nonlocal values are to be converted to *sends* and *receives* by the compiler; finally, synchronization and other control operators must be implemented, though the type architecture does not favor such centralized facilities as semaphores. These language model revisions are so difficult to achieve that one is inclined to recommend definition of a new language. The benefit is that new language facilities can shift some of the burden of allocation, I/O scheduling, synchronization, etc to the programmer and thus deliver efficiently executable programs.

The programmer appears to be poorly served by the CTA. Shared memory might be a thing of the past; and if the new programming languages provide facilities for specifying memory and process allocation, interprocessor data transmission, synchronization, etc, then programmers will have to do all of the detailed planning of the computation. This seems to complicate programming substantially.

In reality, programmers will work harder only to the extent that language designers fail to measure up to the demands of the CTA. For example, shared memory is not excluded by the CTA. Any language designer can choose to provide shared memory as a language abstraction simply by defining it in terms of the CTA—that is, by defining how a compiler should allocate data structures to the processor's local memories, etc. (Architects might help, too, by including automatic routing hardware to support nonlocal references.) This has many benefits, not the least of which is that the language designer, having to explain in the definition of the model how shared memory maps onto a graph structure, will recognize how expensive it can be and thus add language facilities that permit the direct use of the CTA. For example, rather than allow the programmer's encoding of the maximum-finding algorithm to have $O(\log^2 n)$ performance, because of the direct implementation of shared memory, the language designer might add an operator like the APL reduction operation, ⌈/V, which could be implemented by the compiler using the percolate algorithm to achieve $O(\log n)$ performance. Typing three characters to compute

the maximum would not overburden the programmer. The conclusion is that accommodating the programmer's desire for convenience in a CTA-type language depends on the effective use of known facilities and the development of new, more powerful programming abstractions.

Notice that in one sense even a transparent CTA-type language with few abstractions serves the programmers well because they can write efficient programs. After all, it was the distance from the physical machine that earlier prevented the programmer from writing an efficient maximum finding algorithm. In this way the CTA directly serves the need for efficient parallel programming mandated by the Corollary of Modest Potential.

Exploiting Explicit Graph Structure

Return now to the matter of the CTA's unspecified graph structure and recognize that one consequence of the CTA is that every parallel program written in a CTA-type language will be described by a graph. This may be the graph used to define the language model and thus be the same for all programs. On the other hand, if the language is like Poker (Snyder 1984a), each program can use a specific graph. Either way this may be the single greatest benefit of the CTA, because the graph is an explicit statement of the communication structure utilized by the program, and it must be known to implement many (all?) important program optimizations.

The program is not a single graph but rather a family of graphs, where the size of the appropriate graph is determined by the program's input. For example, the SOR runs on a family of square meshes. The program graph, which we call G for *guest,* will have to be mapped by the compiler onto the parallel computer which is also defined in terms of a graph, called H for *host.* The best case is for the graph families to have the same number of vertices, and to be identical (or G contained in H)—that is, the program and the computer are the same graph. Things rarely work out so well.

Assuming the general case where the graph families are different and the size of G (amount of parallelism) greatly exceeds the size of H, there are two problems: contraction and mapping (Berman & Snyder 1984; Bokhari 1981). Contraction is the task of mapping a member of a graph family down to a smaller member of the family; mapping is the task of embedding a graph G into a graph H so that the vertices map one-to-one and the edges of G map to paths of H.

Since there are good contractions for many popular graph families (Berman & Snyder 1984) that can be incorporated into compilers, since there are automatic methods of contraction (Berman et al 1985), and since it is straightforward for programmers to incorporate contractions into their programs (Nelson & Snyder 1986), it seems justified to treat contraction as a solvable problem. (Notice that these contraction methods accomplish the

"downward translation" permitting the program to deliver its rated performance subject to the availability of processors.)

As a result, as long as the program and architecture graphs are selected from the same family, the advertised benefits of the CTA—accurate costs and visible structure—are realized.

When the G family differs from the H graph, the situation is more complicated. Specifically, when G is embedded in H, edges of G map to paths in H, which represents data being relayed through several processors in H to implement what was a direct transfer in G. If the maximum length or dilation (Rosenberg & Snyder 1978) of any path is small, than the cost of running G family programs on H family architectures will be but a constant amount worse than running on the proper architecture. An example would be an eight-degree mesh program run on a four-degree mesh computer. The more typical case is that the worst-case dilation will be larger, on the order of the diameter of the host computer. Thus the shuffle graph might extend the constant transmission time of G to O(log n) time while a mesh architecture would extend it to O($\sqrt{n}$) time. This is a serious departure from what the programmer expects, as serious a deception as when the paracomputer was the type architecture. Moreover, it is a direct consequence of not selecting a specific graph in the CTA, since agreement on one graph would return us to the conjectured-to-be-solvable contraction case above.

So why not simply select a graph and settle the issue? The first answer, as mentioned before, is that no graph has emerged as an optimum choice. To that reason can be added the observation that not all graph structures are equally simple to implement as architectures (Snyder 1984b), and simplicity often translates into a favorable cost-benefit. For example, mesh architectures are very easy in VLSI, but shuffle-based architectures are not. More importantly, no choice has to be made if we learn more about the mapping problem, or learn of ways to cope with it. For example, one might exploit the apparent tendency of graph families to cluster into groups—mesh-based interconnections, cube-based interconnections, etc—and expect problems to be solved for a representative of each group; then the (usually efficient) intratranslatability can be exploited to give an efficient mapping for any member of the group from the base program. Even more attractive would be the development of new, high-level abstractions that have good (but probably different) mappings onto each group. The problem will ultimately be solved, we can be certain; it is simply too early to guess the best solution strategy.

Summary

This critique began with the claim that the medium is the message, that the form of the programming language influences the choice of the algorithms and the details of their encoding into a program. The point was illustrated later

by the poor choice of the maximum-finding algorithm for the paracomputer-based programming language.

Next, various language types were scrutinized: Serial languages were found to be inadequate chiefly because the only available way to specify parallelism is for the compiler to recognize when the given serial order is unimportant. Extended sequential languages were criticized for being unnecessarily constrained by the need to preserve a sequential language model. Parallel languages were seen to pose an almost insurmountable challenge to provide convenient high-level facilities while providing a means to exploit machine-specific features that provide efficiency. This challenge was then placed in perspective; not only is it solvable, but it also promises to be a rich area for further research.

The key to solving the problem was first to appreciate the role of a type architecture and then to choose an effective one. A type architecture is an idealized machine that serves as a region of consensus between programming languages and architectures. The von Neumann architecture is a type architecture, the paracomputer is used as a type architecture, and I here offered a new type architecture called the CTA.

The paracomputer was scrutinized in terms of its role as a type architecture and was shown to be wanting. It failed to meet two critical requirements of a type architecture: to describe costs accurately and to expose important structural information. By misrepresenting costs, the paracomputer leads one to select suboptimal algorithms; and one cannot tell the compiler how to generate optimal code without seeing the cost critical features of the architecture from the language. Finally, dance hall architectures do nothing to reduce the inadequacies of the paracomputer.

In the interest of offering a constructive alternative to the paracomputer, I introduced a candidate type architecture, dubbed the CTA. This non-shared-memory machine avoided the shortcomings of the paracomputer but seemed to introduce its own problems because the graph describing its interconnection structure was left open. It was argued that it is premature to select a graph because none has emerged as a clear choice. Furthermore, programming languages in which the graph is specified not only finesse the issue but also make explicit the actual communication requirements of the program. Finally, shared memory, as an abstraction, can and probably should be defined on top of the CTA; it is a useful programming facility, and there is no reason why the CTA should prevent the programmer's use of it.

The opening observation was the Corollary of Modest Potential, an obvious but not widely appreciated interpretation of the Fundamental Law, stating that the greater a problem's sequential time complexity, the less improvement in terms of increased problem size will be realized by parallelism. This observation motivated an intense interest in avoiding inefficiency. But in most of our

discussions the gap between efficient and inefficient was generally O(log n), an amount sufficiently small that we might be inclined to ignore it. But that conclusion misses the import of the Corollary.

Most of the arguments were intended to establish a gap between efficiency and standard practice; O(log n) was the easiest gap to establish. It is generally not known how inefficient most of the cases discussed can become, many complexity-contributing aspects of parallel computation are routinely ignored, and the critical constants of proportionality are all but unknown. The existence of the gap has generally been the important point; its magnitude could well be larger. Even so, a logarithm is not as small as our common usage tends to imply. When the 100,000,000 processors of the original example, which improved the problem size by only a factor of 100, are handicapped by a logarithm's worth of overhead, more than half of this modest improvement is lost!

ACKNOWLEDGMENTS

It is a pleasure to thank W. L. Ruzzo for many enjoyable hours of illuminating discussion on the topics of this paper; his thoughtful comments on earlier drafts of the manuscript have improved the work immeasurably. It is a pleasure to thank Loyce M. Adams for her patient instruction on the subtleties of successive overrelaxation. Faith E. Fich has been extremely helpful in clarifying these ideas and suggesting the interpretation of Valiant's algorithm. Burton J. Smith offered helpful insight into the impact of practical constraints on parallel type architectures. Kenneth Kennedy, Jean-Loup Baer, Janice Cuny, Duncan Lawrie, and Allan Gottlieb helped with specific technical questions. The Blue CHiPpers generously offered their suggestions to improve the presentation of these ideas. This assistance is greatly appreciated.

Literature Cited

Ackerman, W. B. 1982. Data flow languages. *Computer* 15(2):15–25

Adams, L. M. 1982. *Iterative algorithms for large sparse linear systems on parallel computers*. PhD thesis. Univ. Va., Charlottesville

Adams, L. M., Jordan, H. F. 1986. Is SOR colorblind? *SIAM Sci. Stat. Comput.* 7(2):490–506

Allen, J. R., Kennedy, K., Porterfield, C., Warren, J. 1983. Conversion of control dependencies to data dependencies. *Proc. 10th ACM Symp. Princ. Program. Lang., Austin, Texas*, pp. 177–89

Archibald, J., Baer, J.-L. 1985. An evaluation of cache coherence solutions in shared-bus multiprocessors. *Univ. Wash. Tech. Rep. 85-10-05*

Bardon, M., Curtis, K. K. 1983. *A National Environment for Academic Research*. Washington, DC: Natl. Sci. Found.

Berman, F. D., Goodrich, M., Koelbel, C., Robinson, W. J. III, Showel, K. 1985. Prep-P: a mapping preprocessor for CHiP computers. *Proceedings of the 1985 International Conference on Parallel Process., IEEE*, ed. D. Degroot, pp. 731–33. Washington, DC: Comput. Soc. Press

Berman, F. D., Snyder, L. 1984. On mapping parallel algorithms into parallel architectures. *Proc. 1984 Int. Conf. Parallel Process., IEEE, Belaire, Mich.*, ed. R. Keller, pp. 307–9

Bokhari, S. H. 1981. On the mapping problem. *IEEE Trans. Comput.* C30(3):207–14

Boral, H., DeWitt, D. J. 1983. Database

machines: An idea whose time has passed? A critique of the future of database machines. In *Database Machines*, ed. H. O. Leilich, M. Missikoff, pp. 166–87. New York: Springer-Verlag

Borodin, A., Hopcroft, J. 1982. Routing, merging and sorting on parallel models of computation. *Proc. 14th Ann. Symp. Theory Comput., ACM, San Francisco*, pp. 338–44

Bouknight, W. J., Denenberg, W. J., McIntyre, S. A., Randall, D. E., Slotnick, D. L. 1972. The ILLIAC IV system. *Proc. IEEE* 60(4):369–88

Bromley, K., Symanski, J. J., Speiser, J. M., Whitehouse, H. J. 1981. Systolic array processor developments. In *VLSI Systems and Computations*, ed. H. T. Kung, B. Sproull, G. Steele, pp. 273–84. Rockville, Md: Comput. Sci. Press

Chen, M. C. 1986. A parallel language and its compilation to multiprocessor machines on VLSI. *Proc. 13th ACM Symp. Princ. Program. Lang.* In press

Cook, S. A., Dwork, C., Reischuk, R. 1986. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.* 14(3):688–708

Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., Blackadar, T. 1985. Performance measurements on a 128-node butterfly parallel processor. See Berman et al 1985, pp. 531–40

Dohi, Y., Fisher, A. L., Kung, H. T., Monier, L. M. 1985. The programmable systolic chip: project overview. In *Algorithmically Specialized Parallel Computers*, ed. L. Snyder, L. Jamieson, D. Gannon, H. Siegel, pp. 47–53. New York: Academic

Fielland, G., Rogers, D. 1984. 32-bit computer system shares land equally among up to 12 processors. *Electronic Design*, pp. 153–68

Fisher, J. A. 1983. Very long instruction word architectures and the ELI-512. *Tech. Rep. YALEU/DCS/RR-253, Yale Univ., New Haven, Conn.*

Franklin, M. A., Dhar, S. 1986. On designing interconnection networks for multiprocessors. *Tech. Rep. WUCS-86-3, Washington Univ., St. Louis, Mo.*

Gajski, D., Kuck, D., Lawrie, D., Sameh, A. 1983. Cedar—a large scale multiprocessor. *Proc. 1983 Int. Conf. Parallel Process., IEEE, Belaire, Mich.*, pp. 524–29

Gottlieb, A. 1981. A historical guide to the ultracomputer literature. *Ultracomputer Note #36, New York Univ., NY*

Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M. 1983. The NYU ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Trans. Comput.* C32(2):175–89

Hockney, R. W., Jesshope, C. R. 1981. *Parallel Computers*. Bristol, UK: Adam Hilger

Hwang, K., Briggs, F. A. 1984. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill

INMOS Limited. 1984. *OCCAM Programming Manual*. Englewood Cliffs, NJ: Prentice-Hall

Keller, R. M. 1982. Comment in a technical presentation. *Parallel Architectures Workshop, Boulder, Colo.*

Kuck, D. J., Kuhn, R. H., Leasure, B., Wolfe, M. 1980. The structure of an advanced vectorizer for pipelined processors. *Proc. 4th Int. Comput. Software Appl. Conf., IEEE*, pp. 709–15

Kung, H. T., Leiserson, C. E. 1980. Algorithms for VLSI processor arrays. In *Introduction to VLSI Design Systems*, ed. C. Mead, L. Conway, pp. 271–92. Reading, Mass: Addison-Wesley

Lawrie, D. H. 1975. Access and alignment of data in an array processor. *IEEE Trans. Comput.* C24(12):1145–55

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM* 3(4):184–95

McLuhan, M. 1964. *Understanding Media: The Extensions of Man*. New York: McGraw-Hill

Naur, P., ed. 1963. Revised report in the algorithmic language ALGOL-60. *Commun. ACM* 6(1):1–17

Nelson, P. A., Snyder, L. 1986. Programming solutions to the algorithm contraction problem. *Tech. Rep. #86-03-02, Comput. Sci. Dep., Univ. Wash., Seattle*

Pfister, G. F., Brandlay, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., et al. 1985. The IBM research parallel processor prototype (RP) introduction and architecture. See Berman et al 1985, pp. 764–71

Preparata, F., Vuillemin, J. 1981. The cube-connected cycles: A versatile network for parallel computation. *Commun. ACM* 24(5):300–9

Rosenberg, A. L., Snyder, L. 1978. Bounds on the cost of data encoding. *Math. Syst. Theory* 12:9–39

Schwartz, J. T. 1980. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2(4):484–521

Seitz, C. E. 1985. The cosmic cube. *Commun. ACM* 28(1):22–33

Shapiro, E. 1984. Systems programming in concurrent prolog. In *Proc. 11th ACM Symp. Princ. Program., Salt Lake City, Utah*, pp. 93–105

Shiloach, Y., Vishkin, U. 1981. Finding the maximum, merging and sorting in a parallel

computation model. *J. Algorithms*. 2:88–102

Siegel, H. Jay. 1985. *Interconnection Network for Large-Scale Parallel Processing*. Washington DC: Heath

Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T. Jr., Smalley, H. E. Jr., Smith, S. D. 1981. PASM: A partitionable SIMD/MIMD system for image processing and patter recognition. *IEEE Trans. Comput*. C30(12):934–47

Smith, B. J. 1981. Architecture and applications of the HEP multiprocessor computer system. *Proc. SPIE Symp.*, pp. 242–48

Snir, M. 1985. On parallel searching. *SIAM J. Comput*. 14(3):688–708

Snyder, L. 1982. Introduction to the configurable, highly parallel computer. *Computer* 15(1):47–56

Snyder, L. July 1984a. Parallel programming and the poker programming environment. *Computer* 17(7):27–36

Snyder, L. 1984b. Supercomputers and VLSI: The effect of large-scale integration on computer architecture. In *Advances in Computers*, ed. M. C. Yovitz, 23:1–33. New York: Academic

Snyder, L. 1985. An inquiry into the benefits of multigauge computation. See Berman et al 1985, pp. 488–92

Su, W.-K., Faucette, R., Seitz, C. 1985. C programmer's guide to the cosmic cube. *Tech. Rep. 5203: TR: 85, Comput. Sci. Dep., Calif. Inst. Technol., Pasadena, Calif.*

Upfal, E., Wigderson, A. 1984. How to share memory in a distributed system. *Proc. 25th IEEE Symp. Found. Comput. Sci.*, pp. 171–80

Valiant, L. G. 1975. Parallelism in comparison problems. *SIAM J. Comput*. 4(3):348–55

Valiant, L. G., Brebner, G. J. 1981. Universal schemes for parallel communication. *Proc. 13th ACM Symp. Theory Comput., Milwaukee, Wisc.*, pp. 263–77

Wulf, W. A., Bell, C. G. 1972. C.mmp—A multi-miniprocessor. *Proceedings AFIPS Fall Joint Computer Conference*, 41:765–77. Montvale, N.J: AFIPS Press

Young, D. 1950. *Iterative methods for solving partial differential equations of elliptic type*. PhD thesis. Harvard Univ., Cambridge, Mass.

Young, D. 1971. *Iterative Solution of Large, Linear Systems*. New York: Academic