The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 58

ON THE POWER OF PARALLEL MACHINES WITH HIGH-ARITY INSTRUCTION SETS

BY

IAN PARBERRY

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
England.

November 1983
(updated Feb 1984)

On the power of parallel machines with high-arity instruction sets

Ian Parberry

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL,
England.

ABSTRACT

We consider various models of parallel computers based on communication networks of sequential processors. The *degree* of a parallel machine is the number of communication lines connected to each processor; the *arity* is the number of these lines which a processor can actively manipulate at any given time. The emphasis of current research has been placed on constant-arity machines; however, machines with higher arity have occasionally made an appearance.

Our aim is to investigate the relative computing power of these high-arity parallel computers. We present a high-arity model and show that machines with arity and degree A(n) are more powerful than those of arity o(A(n)). Despite this, we are able to show that P(n) processor parallel computers with arity and degree O(log P(n)) are not much more powerful than those of constant degree, in the sense that they can be efficiently simulated by a practical universal parallel machine.

On the power of parallel machines with high-arity instruction sets

Ian Parberry

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL,
England.

1. Introduction.

In this paper we investigate the relative computing power of some similar parallel machine models. Very loosely, a parallel computer consists of an infinite family of finite graphs, one for each input size. Each vertex represents a processor, and each edge a communication line between processors. The *degree* of a parallel machine is the degree of its interconnection graph, as a function of input size. A communication line between processors A and B is made up of two bidirectional links, one under the control of A and the other under the control of B. We call a processor's links *active* if they are under its control, and *passive* otherwise. The models considered here will differ in the precise details of how a processor is to manage its active links. This paper contains a preliminary version of results to appear in the author's Ph. D. thesis [23]. The reader requiring a more detailed account is directed to that reference.

We make the reasonable assumption that the average user would be unable or unwilling to fabricate a special-purpose parallel machine for each application, and would instead prefer to pay the cost of a moderate increase in resources in return for the ability to use standard components. In order to make this a practical proposition, it is sufficient to demonstrate the existence of a universal parallel machine which is particularly easy to build (for example, its interconnection graph should have constant degree and be easy to construct), and can

efficiently simulate any reasonable parallel machine.

A number of universal parallel machines of this nature have already received some attention in the current literature (for a survey, see [23]). It can be shown [14, 19] that there exists a P(n) processor universal parallel computer which can simulate any P(n) processor machine with a cost of $O(\log^2 P(n))$ steps for each step of the simulation. It can also be shown [14] that this cost can be reduced to $O(\log P(n))$ if the machine being simulated has constant degree. This can be taken as an indication that constant-degree parallel computers are less powerful than those of arbitrary degree. It should be noted that the cost can also be reduced to $O(\log P(n))$ by substituting the sorting algorithm of [2] in the construction of [14, 19], although the constant multiple in the asymptotic bound is too large to be of any practical use.

These results assume strict limitations on the way in which the active and passive links of any individual processor may be used. In [14, 19], during any given time-step, for any given processor:

(1) Only one active link may be used.

(2) All passive links which are transmitting data must carry the same value.

(3) Only one of the passive links which are attempting to deliver data is allowed to succeed; all other incoming passive data is lost.

Conditions (2) and (3) are relaxed in the "network machines" of [23], leading to slightly different resource bounds in the universal machine.

We extend this model to give each processor the use of more than one active link simultaneously (and the power to make efficient use of the data thus obtained), whilst maintaining conditions (2) and (3). In addition, these latter conditions may be relaxed using the techniques of [23]. We call the number of active links which can be used in any time-step the *arity* of a parallel machine.

We show that machines of arity and degree A(n) are more powerful than those of arity o(A(n)). In particular we are interested in parallel machines with reasonably small arity and degree; more precisely, those with P(n) processors and arity and degree O(log P(n)). Whilst it is apparent from the above that these machines are more powerful than those of constant degree, we can show that they are not too much more powerful, in the following sense. There is a universal machine (with interconnections based on the cube-connected-cycles [25] or shuffle-exchange [30]) with O(P(n).log P(n)) processors which can simulate any P(n) processor, degree and arity O(log P(n)) parallel machine at a cost of only O(log P(n)) steps for each simulated time-step. This compares favourably with the corresponding result for the simulation of constant-degree machines.

Although machines with non-constant arity have already appeared in the literature, there has so far been no systematic investigation into the extra computing power offered by high-arity instructions. For example, the random-routing results of [33, 34] initially appeared in a high-arity form, although this has since been redressed in [3, 5, 26, 31]. The oblivious lower-bound of [6] is presented for high-arity machines.

The structure of this paper is as follows. The main body is divided into three sections. In the first section we introduce our basic machine model. In the second we extend it to allow high-arity processors, and prove some asymptotic upper and lower bounds on the running-time of some high-degree and high-arity computations. In the final section we present our universal machine.

2. A Model Of Parallel Computation.

Our parallel machine model consists of an infinite number of synchronous random-access machines (only finitely many of which become involved in any particular computation). By "random-access machine" we refer to a variant of the RAM (which is already well-known as a sequential machine model, see for

example, [1]). By "synchronous" we mean that the instruction-cycles of the RAMs are synchronized.

Each RAM has a number of user-accessible registers. These include an infinite number of general-purpose registers $r_0, r_1, \ldots$ and a communication register COM. The communication register is the only register which is accessible to other processors. It also possesses a number of registers which are preset at the beginning of the computation. These correspond to values which are "hard-wired" into the machine during the fabrication process. They include the processor identity register PID (which is preset to i in the i^{th} RAM, $i = 0,1,\ldots$), the size and degree registers SIZE, DEGREE and an infinite number of port registers $p_0, p_1, \ldots$. All registers are capable of holding a single integer.

More formally, a parallel machine consists of a program P and an interconnection scheme S. The program is a finite list of instructions, each of which have the following form (where p is a port register). Either:

(1) Read the communication register of processor p.

(2) Write to the communication register of processor p.

(3) Perform an internal computation.

(4) Conditional transfer of control, or halt.

For example, let "~" denote any two-input boolean function, addition, subtraction, multiplication or integer division. For convenience, we divide our example instruction-set into two categories. Local instructions have the form:

$r_i \leftarrow$ constant	(load register with constant)
$r_i \leftarrow r_j \sim r_k$	(binary operation)
$r_i \leftarrow r_{r_j}$	(indirect load)
$r_{r_i} \leftarrow r_j$	(indirect store)
$r_i \leftarrow$ PID	(store processor identity)
halt	(end execution)
goto m **if** $r_i > 0$	(conditional transfer of control)

Communication instructions are of the form:

$COM(r_i) \leftarrow r_j$ (write to processor p_{r_i})
$r_i \leftarrow COM(r_j)$ (read from processor p_{r_j})

P represents a program to be run on all active processors in the infinite network. Only a finite number of processors will be active during any computation. Each processor has the power to become inactive by using the "halt" instruction at its own discretion. All active processors synchronously execute *P*; their behaviour is that of independent RAMs, except where communication instructions are concerned. Each processor interprets references to registers in local instructions as references to its own local registers.

The execution of an instruction $r_j \leftarrow COM(r_k)$ by processor i has the effect of reading the communication register of processor p_{r_k} and placing the result into register r_j of processor i. Execution of $COM(r_j) \leftarrow r_k$ has the effect of writing the contents of register r_k of processor i into the communication register of processor p_{r_j}. We assume some reasonable protocol for dealing with multiple attempts to read from or write to the same processor, much in the manner of the popular parallel machine models that use a globally-accessible memory for interprocessor communication (see, for example, [6, 8, 12, 13, 14, 16, 18, 28, 29, 32, 33, 35, 36]). For definiteness, we allow multiple reads, and in the case of write-conflicts, the lowest-numbered processor wins. Attempts to write to the communication register of an inactive processor have no effect; attempts to read it return zero.

Let **Z** denote the set of integers and **N** the set of non-negative integers. An *interconnection scheme S* consists of three functions, a processor function $P:\mathbf{N}\rightarrow\mathbf{N}$, a degree function $D:\mathbf{N}\rightarrow\mathbf{N}$, and an interconnection function

$$G:\{i \mid 0\leq i<P(n)\}\times\{d \mid 0\leq d<D(n)\}\times\mathbf{N}\rightarrow\{i \mid 0\leq i<P(n)\}.$$

In a P(n) processor computation, processor i is connected to processors G(i,d,n), $0\leq d<D(n)$. We adopt the convention that if $i\in\{G(j,d,n) \mid 0\leq d<D(n)\}$ then

$j \in \{ G(i,d,n) \mid 0 \leq d < D(n) \}$.

Suppose $f:\mathbf{N}^* \rightarrow \mathbf{N}^*$, and $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ where $x_i \in \mathbf{N}$ for $0 \leq i <$ n ; we say that x has *size* n, and write $|x| = n$. Let $M = (P,S)$ be a parallel machine, where $S = (P,D,G)$. We define the function computed by M as follows. For $0 \leq i < P(n)$ do the following. Place x_i into register r_0 of processor i, set all other general-purpose and all communication registers to zero. Set SIZE of processor i to P(n) and DEGREE to D(n). For $0 \leq d < D(n)$ set register p_d of processor i to G(i,d,n), and for $d \geq D(n)$ set register p_d of processor i to P(n). Now simultaneously activate processors i with $0 \leq i < P(n)$ on program P. Let y_j be the contents of register r_0 of processor j upon termination, $0 \leq j < |f(x)|$, and $y(x) = \langle y_0, y_1, \ldots, y_{|f(x)|-1} \rangle$. We assume, of course, that $P(n) \geq |x|$, $|f(x)|$, and adopt the convention that $y_j = 0$ for $|f(x)| \leq y < P(n)$. We say that M *computes* f if for all inputs x, y(x) = f(x).

Such a parallel machine is said to have P(n) *processors* and *degree* D(n). It is also said to compute within *time* T(n) for $T:\mathbf{N} \rightarrow \mathbf{N}$ if for all $n \geq 0$, every processor halts within T(n) steps. We will be concerned primarily with these three resource bounds. Other possible resources of interest include *space*, a measure of the number of registers used (summed over all processors, see for example [23]), and *word-size*. A parallel machine has *word-size* W(n) if all values placed into a register during a computation on an input of size n have absolute value at most $2^{W(n)}$.

The astute reader will have noticed that by choosing the unit-cost measure of time, and including multiplication in the instruction-set of our processors, we have allowed our RAMs to become as powerful as parallel machines [17]. Whilst this approach has some theoretical interest [29], we prefer to use purely sequential processors. The problem is apparently not due to the intrinsic power of multiplication, but rather the fact that it can be used to generate very long

integers in a small amount of time [26]. For the purpose of obtaining upper bounds, we will concentrate on machines with $W(n) = T(n)^{O(1)}$. If the internal computations are simple enough, this is sufficient to ensure that our parallel machines obey the parallel computation thesis [8, 16]. If (space.wordsize) is used as a "hardware" measure then this also ensures [23] that they obey the extended parallel computation thesis [10, 11]. The number of processors is also a valid "hardware" measure, provided $T(n) = P(n)^{O(1)}$. Our universal parallel machines will increase space and wordsize by at most a constant multiple.

Instead of writing algorithms in the low-level RAM language, we will follow the common practice of using a high-level language which can easily be translated into instructions of this form. We use the usual high-level constructs for flow-of-control, based on sequencing, selection and iteration. Variables of the form (x **of processor** i) will be taken as a reference to variable x of processor i. An unmodified variable x will be taken to mean (x **of processor** PID), i.e. a local variable. For example, the statement

if y < (y **of processor** $\lfloor PID/2 \rfloor$)
then $statement_1$
else $statement_2$

causes the i^{th} processor, $0 \leq i < P(n)$, to (provided it is still active) simultaneously compare its variable y with variable y of processor $\lfloor i/2 \rfloor$. If the former is less than the latter, then $statement_1$ is executed, otherwise $statement_2$. To aid synchronization, we assume that the code generated for $statement_1$ and $statement_2$ has the same number of instructions, by filling with NO-OPs (such as $r_0 \leftarrow r_0$) as necessary.

Although every processor of our parallel machine executes the same program, our model does not fall precisely into the SIMD category of Flynn [12]. This is because the conditional goto statement takes action depending on the

value of a local register, the contents of which may vary from processor to processor. Thus different processors may be at different points in the program at any given time. However, it is fairly easy to show [25] that our model is equal in power to a SIMD one, and to a reasonable subset of MIMD models, including that of Galil and Paul [14].

3. A High-Arity Model.

In this section we generalize our parallel machine model to give each processor the power to communicate with asymptotically more than a constant number of its neighbours in unit time; and sufficient power to make efficient use of the information thus obtained. Let $A:\mathbf{N}\rightarrow\mathbf{N}$. We allow our processors to have instructions which can be simulated in time $O(A(n))$ by the processors of section 2, where n is the input size. $A(n)$ is then called the *arity* of the machine. For example, we replace the instruction-set of section 2 with the following instruction-set:

(1) $r_i[r_j \leftarrow \text{constant}]$ (block-load constant)
(2) $r_i[r_j \leftarrow r_k]$ (duplicate register)
(3) $r_i[r_j \leftarrow r_k \sim r_l]$ (element-by-element operation)
(4) $r_i[r_j \leftarrow \sim r_k]$ (prefix $\sim$, where $\sim$ is associative)
(5) $r_i[r_j \leftarrow r_{r_k}]$ (indirect loads)
(6) $r_i[r_{r_j} \leftarrow r_k]$ (indirect stores)
(7) $r_j \leftarrow \text{PID}$
(8) halt
(9) **goto** m **if** $r_j > 0$
(10) $r_i[\text{COM}(r_j) \leftarrow r_k]$ (write into COM registers)
(11) $r_i[r_j \leftarrow \text{COM}(r_k)]$ (read from COM registers)

Instructions (7-9) are as in section 2. Instructions (1-3,5,6,10,11) have the same effect (in unit time) as the high-level statement:

for m:=0 **to** r_i-1 **do** S

where statement S is respectively

(1) $r_{j+m}:=$constant,

(2) $r_{j+m}:=r_k$,

(3) $r_{j+m}:=r_{k+m}{}^{\sim}r_{l+m}$,

(5) $r_{j+m}:=r_{r_{k+m}}$,

(6) $r_{r_{j+m}}:=r_{k+m}$,

(10) $\mathrm{COM}(p_{r_{j+m}}):=r_{k+m}$

(11) $r_{j+m}:=\mathrm{COM}(p_{r_{k+m}})$.

Instruction (4) has the same effect as

$r_j:=r_k$
for $m:=1$ **to** r_i-1 **do**
$r_{j+m}:=r_{j+m-1}{}^{\sim}r_{k+m}$

In this particular model, a parallel machine has arity A(n) if for all inputs of size $n\geq 0$, the largest value present in register r_i during the execution of instructions of the form (1-6,10,11) is at most A(n). Whilst high-arity (i.e. $A(n)=\omega(1)$) parallel machines make an interesting vehicle for theoretical study, they are perhaps too powerful for practical purposes. In this case one should restrict oneself to parallel machines which obey the parallel computation thesis [8, 16] and the extended parallel computation thesis [10, 11]. In order to achieve this, it would be reasonable to consider only parallel machines for which there exists a constant-arity equivalent with only a polynomial increase in the other resource bounds (as is the case when $A(n)=T(n)^{O(1)}$).

Some of the power of high-arity machines comes from the fact that they have high degree. It is easy to show that a machine with degree D(n) is asymptotically faster that any machine of degree o(D(n)). Consider the problem of broadcasting a single value amongst n processors. More formally, we wish to compute, in parallel, the function $f:\mathbf{N}^*\rightarrow\mathbf{N}^*$ defined by

$f(x_0, \ldots, x_{n-1}) = (y_0, \ldots, y_{n-1})$ where $y_i = x_0$ for $0 \le i < n$. Suppose $d \ge 3$. The following is an n processor, degree d algorithm for computing f on inputs of size n in time $O(\frac{\log n}{\log d})$. Assuming that initially variable x of processor i contains x_i, the algorithm terminates with variable x of processor i containing x_0, $0 \le i < P(n)$. The interconnection pattern used is a (d-1)-ary tree.

```
b:=1
while b < n do
  x:=x of processor ⌊(PID − 1)/(d − 1)⌋
  b:=b.(d−1)
```

The time-bound attained by the above algorithm is asymptotically optimal. It is easy to see that a degree d machine must take time $\Omega(\frac{\log n}{\log d})$ to compute f, regardless of how complicated its interconnections are, how many processors are used, or what the arity of those processors is. This follows from the observation that there must be a path in the interconnection graph of size n from processor 0 to processor i, $0 \le i < P(n)$. From this we can conclude that a degree D(n), constant arity parallel machine with n processors is asymptotically faster than than any machine of degree o(D(n)). Indeed, the latter machine may even be allowed to have a different non-recursive program for each processor, which may vary with input size.

Furthermore, high arity parallel machines are even more powerful than high degree ones. We will show that a parallel machine with arity and degree D(n) is asymptotically faster than any machine with arity o(D(n)). Consider the problem of computing the sum of n integers. More formally define $sum: \mathbf{N}^* \to \mathbf{N}$ by

$$sum(x_0, \ldots, x_{n-1}) = \sum_{i=0}^{n-1} x_i.$$

Suppose $d \ge 2$. The following is an n processor algorithm with degree d+1 and

arity d for computing the sum of n integers in time $O(\frac{\log n}{\log d})$. Initially, we assume that variable x of processor i contains x_i, $0 \le i < P(n)$. On termination, variable s of processor 0 contains $sum(x_0, \ldots, x_{n-1})$. The interconnection pattern is a d-ary tree.

```
s,b:=0,1
while b < n do
  s:=x+ Σ_{i=PID.d+1}^{PID.d+d} (s of processor i)
  b:=b.d
if PID > 0 then s:=0
```

Thus for $D(n) \ge 2$, an n processor, arity D(n), degree D(n)+1 parallel machine can compute the sum of n integers in time $O(\frac{\log n}{\log D(n)})$. This algorithm is asymptotically optimal for all machines of arity O(D(n)). In fact, we will show that an arity D(n) machine must take at least $\left\lfloor \frac{\log n}{\log (D(n)+2)} \right\rfloor$ steps to sum n integers.

Suppose M is a P(n) processor parallel machine of arity D(n) which can sum n integers in time T(n), and let $x = \langle x_0, \ldots, x_{n-1} \rangle$ be an input string consisting of n symbols, each of which is a non-negative integer. Let G_x be the directed graph with vertices (p,t), $0 \le p < P(n)$, $0 \le t < T(n)$, and an edge from (p_1,t_1) to (p_2,t_2) if $t_2 = t_1+1$ and either $p_1 = p_2$ or during time-step t_1 of the computation of M on input x, either processor p_2 reads a value from p_1, or p_1 successfully writes a value to p_2. The i^{th} symbol of x is said to be *reachable* if there is a path from vertex (i,0) to vertex (0,T(n)) in G_x. The *reachable string* is the string derived from x by deleting all unreachable symbols. The *unreachable string* is similarly derived by deleting all reachable symbols.

Suppose the values to be added together are all less than N. We claim that (provided N is sufficiently large) there is an input string in which all symbols are

reachable. For a contradiction, suppose that every input has at least one unreachable symbol. Fix a graph G_x and consider the strings y such that $G_x = G_y$. Each reachable symbol of y can take on N possible values, giving a total of N^r possible reachable strings, where $r<n$ is the number of reachable symbols. Further, for each reachable string, the corresponding unreachable string must sum to a fixed value, dependent only on the reachable string in question. This follows because M must give the same result for two inputs y_1, y_2 such that $G_{y_1} = G_{y_2}$ and y_1, y_2 have identical reachable strings. Since $m \geq 1$ non-negative integers can sum to a fixed value at most N^{m-1} times, we see that there are at most N^{n-r-1} unreachable strings which can appear with any reachable string, and thus there are at most N^{n-1} choices of y. That is, each graph G_x can be used for at most N^{n-1} different input strings x.

Let $G(n) = |\{G_x | x \in N^n\}|$. By the pigeonhole principle, at least one graph must be used for at least $N^n / G(n)$ input strings. If N is chosen such that $N > G(n)$ then this value is greater than N^{n-1}, which contradicts the result of the previous paragraph. Thus there must be an input string for which all symbols are reachable. Since for all x, G_x has in-degree $D(n)+2$, this implies that $T(n) \geq \left\lceil \frac{\log n}{\log (D(n)+2)} \right\rceil$.

Unfortunately, this proof is based heavily on the use of extremely large integers as summands. Indeed, it may be necessary to choose N to be as large as $P(n)^{(D(n)+1).P(n).T(n)}$. Thus:

(1) If we insist that $W(n) = T(n)^{O(1)}$ (which, as we saw in section 2, ensures that the parallel computation thesis holds), then the lower-bound is not valid.

(2) For machines with $W(n) = n^{O(1)}$ (which is a reasonable restriction since it ensures that the input encoding is "concise" in the sense of [15]), the lower-bound holds provided $P(n) = n^{O(1)}$.

(3) If the word-size is arbitrary, the lower-bound holds regardless of arity or number of processors. This is despite the fact that machines with $W(n) = 2^{O(T(n))}$ are (as observed in [6]) exceptionally powerful.

4. Universal Machines and Small Arity.

In this section we consider parallel machines with small degree and arity. By "small" we mean P(n) processor machines with degree and arity O(log P(n)). From section 3 we know that machines with small degree and arity are more powerful than machines with constant degree. In this section we propose to show that they are not much more powerful.

Suppose $M = (P,S)$ is a parallel machine, where $S = (P,D,G)$. A simulation of M on an input of size n by a universal machine U is to proceed as follows. Processor i of U, $0 \le i < P(n)$ is presented with a description of P, P(n), D(n) and G(i,d,n) for $0 \le d < D(n)$. The SIZE, DEGREE and port registers of U are initialized as described in section 2, and the input of M is presented to U in the same manner. A sufficient number of processors of U are activated. The *delay* of a simulation is the time to simulate a single step of M, and the *setup time* the time to initialize U before the simulation proper begins.

We present our proof-sketch using a multi-dimensional cube as the interconnection pattern. The reader familiar with the work of Preparata and Vuillemin [27] will have little difficulty in implementing our algorithm on either the cube-connected-cycles [27] or shuffle-exchange [32] interconnection patterns. If l is a non-negative integer with binary representation $l_{k-1} \cdots l_1 l_0$ $(k \ge \lceil \log_2(l+1) \rceil)$, let $l^{(i)}$ denote the integer which differs from l in bit l_i, that is, $l^{(i)} = l + (-1)^{l_i}.2^i$. The *k-cube*, $k \ge 0$, has vertex-set $\{l | 0 \le l < 2^k\}$ and each vertex l is joined to vertices $l^{(i)}$ for $0 \le i < k$.

We will make use of the following sub-algorithms.

Algorithm 1. Suppose processor 0 has a value x which it wishes to broadcast to processors $0,1,\ldots,2^k-1$. This can be achieved by using a simple-ASCEND class algorithm (see [27] for terminology). The following algorithm terminates with variable v of every processor equal to x.

```
v:= if PID=0 then x else 0
for b:=0 to k−1 do
  if v=0 then v:=(v of processor PID^(b))
```

Algorithm 2. Suppose processor 0 has 2^k items of data in an array. We wish to scatter these amongst processors $0,1,\ldots,2^k-1$ in such a manner that each processor receives precisely one value. The algorithm consists of k stages. At the start of stage i, $0 \le i < k$, the 2^i processors p with $0 \le p < 2^i$ are each in possession of 2^{k-i} data items. Stage i consists of processor p, $0 \le p < 2^i$ sending 2^{k-i-1} of its data items to processor $p^{(i)}$. In the following algorithm, processor 0 starts off with 2^k items of data in array $d[0..2^k-1]$. Each processor receives its value into variable d[0].

```
for i:=0 to k−1 do
  for j:=0 to 2^(k−1−i)−1 do
    if PID_i=1 then d[j]:=(d[j+2^(k−1−i)] of processor PID^(i))
```

Algorithm 3. Suppose each processor i, $0 \le i < 2^k$ holds an integer value in a variable l. The *local rank* of processor i is defined to be $|\{ j \mid 0 \le j < i \text{ and } (l \text{ of processor } j) = (l \text{ of processor } p) \text{ for } j < p \le i\}|$. The following simple ASCEND class algorithm sets variable R of each processor to its local rank.

```
S,V,R:=1,1,0
for b:=0 to k−1 do
 if PID_b = 1
   then if (V of processor PID^(b)) = 1
          then R:=R+(S of processor PID^(b))
   else if (V of processor PID^(b)) ≠ V
          then S:=0
        V:=(V of processor PID^(b))
  S:=S+(S of processor PID^(b))
```

If $0 \le i < 2^k$, define the *b-block* (after [19]) of processor i to be the set of 2^b processors $\{ \lfloor i/2^b \rfloor . 2^b + j \mid 0 \le j < 2^b \}$. At the start of the b^{th} iteration of the for loop, $0 \le b < k$, variable R of processor i, $0 \le i < 2^k$ contains that processors rank within its b-block, whilst variables V and S of processor i contain the value of 1 and local rank of processor $\lfloor i/2^b \rfloor . 2^b + 2^b - 1$, (i.e. the last processor in its b-block) respectively. The correctness of the algorithm follows by induction.

Algorithms 1 and 3 run in time O(k) on a k-cube. Algorithm 2 runs in time

$$\sum_{b=0}^{k-1} c.2^{k-1-b}$$

(for some constant c), i.e. $O(2^k)$.

Theorem 1. There is a P(n).D(n) processor universal parallel machine which can simulate any P(n) processor machine of arity-and-degree D(n), with delay $O(\log P(n) + D(n))$ and setup time $O(\log^4 P(n) + D(n))$.

Proof. (Outline). Suppose $m = \lceil \log P(n) \rceil$ and $m' = \lceil \log D(n) \rceil$. We describe our algorithm on an $(m+m')$-cube. Let M be a P(n) processor parallel machine with degree and arity D(n). Processor i, $0 \le i < P(n)$ of the universal machine will simulate processor i of M. Let i[d] denote the d^{th} neighbour of processor i of M, in order of ascending PID. For $0 \le i < P(n)$ let W_i be the m'-cube consisting of processors $2^m.k + i$, for $0 \le k < 2^{m'}$, of the universal machine.

As part of the initialization, each processor i $0 \le i < P(n)$ will receive D(n) identities $I^i_{i[d]}$ for $0 \le d < D(n)$ such that

(1) $0 \le I^i_{i[d]} < D(n)$ for all $0 \le i < P(n)$, $0 \le d < D(n)$, and

(2) For all $0 \le i,i',j < P(n)$, if I^i_j and $I^{i'}_j$ are both defined, and $I^i_j = I^{i'}_j$ then $i = i'$.

Thus processor i is the $(I^i_{i[d]})^{th}$ neighbour of processor i[d], in ascending order of PID.

This is achieved as follows. Processor i $0 \le i < P(n)$ prepares D(n) packets (i[d],i), $0 \le d < D(n)$, and scatters them around the $2^{m'} \ge D(n)$ processors of W_i, at most one packet per processor, using algorithm 2. These packets are then sorted within the $(m+m')$-cube in lexicographic (first-field-first) order. Each processor j, $0 \le j < 2^{m+m'}$ thus receives some packet $(i_j[d],i_j)$. It then sets l_j to $i_j[d]$. Running algorithm 3 on the $(m+m')$-cube computes the local rank of each processor, which in this case is $I^{i_j}_{i_j[d]}$. Armed with this information, for $0 \le i < P(n)$ the processor in charge of packet (i[d],i) transforms it into $(i,i[d],I^i_{i[d]})$. These packets are sorted back to their respective W_i's, and then gathered back into processor i by reversing the scattering algorithm 2.

After the initialization phase, each step of the simulation proceeds as follows. First, requests to read communication registers are fulfilled. Processor i $0 \le i < P(n)$ prepares D(n) request packets $(i[d],I^i_{i[d]},i)$, $0 \le d < D(n)$. These are scattered at most one-per-processor around the processors of W_i, using algorithm 2. Once this has been carried out, let Π be the permutation which carries packet $(i[d],I^i_{i[d]},i)$ to processor $2^m.I^i_{i[d]} + i[d]$, for $0 \le i < P(n)$. Once Π has been applied, W_i contains the D(n) requests from the neighbours of processor i of M, $0 \le i < P(n)$. Processor i can then fulfil the D(n) requests by broadcasting the contents of the communication register of processor i of M around the processors of W_i using algorithm 1. The fulfilled requests are routed back to their originating processors by reversing the above process. Processor i of the universal

machine can then simulate the internal computation of processor i of M, $0 \leq i < P(n)$. Finally, requests to write to communication registers are handled in a similar manner.

Repeating this t times enables us to simulate t steps of M. Note that Π is the same for each step. It is well-known [27, 30] that a fixed permutation can be carried out in time $O(\log P(n))$ by simulating one of Waksman's [39] permutation networks. This requires $O(\log^4 P(n))$ setup time, however [18, 21, 22, 30]. The total setup time is thus comprised of

(1) $O(\log^2 P(n) + D(n))$ to compute the identities $I^i_{i[d]}$. The $\log^2$ term comes from sorting using a straightforward simulation (see, for example, [27, 30]) of one of Batcher's [4] sorting networks. The D(n) term comes from the use of algorithm 2 to scatter D(n) values.

(2) $O(\log^4 P(n))$ to set up Π.

The delay is comprised of

(1) O(D(n)) to prepare and broadcast the request packets,

(2) O(log P(n)) to compute Π,

(3) O(log D(n)) to fulfil the request packets,

which gives us the required result.▪

Corollary. There is an O(P(n).log P(n)) processor universal parallel machine which can simulate, with delay O(log P(n)), any P(n) processor parallel machine of arity and degree O(log P(n)).

The proof of theorem 1 can be modified slightly to give

Theorem 2. There is a P(n).D(n) processor universal machine which can simulate any P(n) processor, degree D(n), arity-1 machine, with delay O(log P(n)) and setup time $O(\log^4 P(n) + D(n))$.

Note that by making D(n) constant in theorem 2, and using the processor-saving theorems of [24] we obtain a shorter proof of theorem 8 of [14] (a P(n) processor universal machine which can simulate any P(n) processor, constant degree parallel machine with delay O(log P(n))).

For high-arity machines of arbitrary degree we have:

Theorem 3. There is an A(n).P(n) processor universal parallel machine which can simulate, with delay $O(A(n)+\log^2 P(n))$, any P(n) processor machine of arity A(n).

Proof. By substituting the sorting algorithm of Batcher [4] (see, for example, [27, 30, 32]) for the permutation Π of theorem 1. ▪

Finally, we should note that theorems 1 and 3 can be improved asymptotically by the use of the sorting algorithm of [2].

Theorem 4. There is an A(n).P(n) processor universal parallel machine which can simulate, with delay O(A(n)+log P(n)), any P(n) processor parallel machine of arity A(n).

Proof. Similar to theorem 3, substituting the sorting algorithm of [2] for that of Batcher. The processor-bound is achieved by use of the pipelining technique of Vishkin [37]. ▪

However, as we observed in the introduction, the constant multiple in the asymptotic time-bound is too large to be of any practical use.

5. Conclusion.

We have seen that high-arity computers with minimal degree are more powerful than those with asymptotically smaller arity. Thus P(n) processor parallel machines with degree and arity O(log P(n)) are potentially much more powerful than those of constant degree. It takes O(log P(n)) steps to simulate a single step of an arbitrary constant-degree machine on a practical universal

machine; yet we can simulate a single step of an arbitrary machine with arity and degree O(log P(n)) in asymptotically the same amount of time, by using only O(P(n).log P(n)) processors. Thus we see that, in a practical setting, machines with reasonably small arity and degree are not much more powerful than their constant-degree counterparts.

The study of universal parallel machines has become a popular pastime. Only the most restrictive models (with, for example, constant degree and very simple interconnection pattern) can be considered practical, yet there are practical universal machines which can simulate even the most impractical (for example, completely-connected) models. This is important for both the theoretician and the more practically inclined. It provides the theoretician with a practical motivation for studying the more abstract parallel machine models (which are often more amenable to formal analysis). In addition, it also gives the user of a practical parallel machine a flexible high-level programming language. This programming language affords (at a small cost in resources) a virtual architecture corresponding to an elegant abstract machine, thus protecting the user from the strict technological constraints governing practical architectures.

Thus we have shown that it is practical to consider parallel machines with non-constant arity (subject to certain reasonable constraints). We have also given some indication of the computing power to be gained by the use of high-arity instruction sets. Finally, we have provided the user of a practical parallel computer with a new, more powerful programming language.

6. Acknowledgements.

I would like to thank Mike Paterson for his comments and guidance during all stages of this work.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley (1974).

2. M. Ajtai, J. Komlós, and E. Szemeredi, "An O(n.log n) sorting network," *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, (Apr. 1983).

3. R. Aleliunas, "Randomized parallel communication," *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, (August 1982).

4. K. E. Batcher, "Sorting networks and their applications," *Proceedings AFIPS Spring Joint Computer Conference* **32** pp. 307-314 (April 1968).

5. P. Beame, "Random routing in constant-degree networks," Technical Report 161/82, Dept. of Computer Science, University of Toronto (1982).

6. N. Blum, "A note on the 'parallel computation thesis'," *Information Processing Letters* **17** pp. 203-205 (1983).

7. A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, (May 1982).

8. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation," *Journal of the ACM* **28**(1)(Jan. 1981).

9. S. A. Cook, "Towards a complexity theory of synchronous parallel computation," *L'Enseignement Mathematique* **30**(1980).

10. P. W. Dymond, "Simultaneous resource bounds and parallel computations," Ph. D thesis, issued as Technical Report TR145/80, Dept. of Computer Science, University of Toronto (Aug. 1980).

11. P. W. Dymond and S. A. Cook, "Hardware complexity and parallel computation," *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, (Oct. 1980).

12. M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE* **54** pp. 1901-1909 (Dec. 1966).

13. S. Fortune and J. Wyllie, "Parallelism in random access machines," *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pp. 114-118 (1978).

14. Z. Galil and W. J. Paul, "An efficient general purpose parallel computer," *Journal of the ACM* **30**(2) pp. 360-387 (Apr. 1983).

15. M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman (1979).

16. L. M. Goldschlager, "A universal interconnection pattern for parallel computers," *Journal of the ACM* **29**(4) pp. 1073-1086 (Oct. 1982).

17. J. Hartmanis and J. Simon, "On the power of multiplication in random access machines," *Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory*, pp. 13-23 (1974).

18. G. F. Lev, N. Pippenger, and L. G. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Transactions on Computers* **C-30**(2)(Feb. 1981).

19. D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Transactions on Computers* **C-30**(2) pp. 101-106 (Feb. 1981).

20. D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," *Journal of the ACM* **29**(3) pp. 642-667 (July 1982).

21. D. Nassimi and S. Sahni, "Parallel algorithms to set up the Benes permutation network," *IEEE Transactions on Computers* **C-31**(2)(Feb. 1982).

22. D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangable switching networks," *Bell Systems Technical Journal* **50** pp. 1579-1618 (1971).

23. I. Parberry, "Some practical simulations of impractical parallel computers," Theory of Computation Report No. 58, Dept. of Computer Science, University of Warwick (December 1983).

24. I. Parberry, "Some processor-saving theorems for synchronous parallel computers," Theory of Computation Report No. 53, Dept. of Computer Science, University of Warwick (October 1983).

25. I. Parberry, "A complexity theory of parallel computation," *Ph. D thesis*, Dept. of Computer Science, University of Warwick, (In preparation).

26. V. Pratt and L. J. Stockmeyer, "A characterization of the power of vector machines," *Journal of Computer and System Sciences* **12** pp. 198-221 (1976).

27. F. P. Preparata and J. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation," *Communications of the ACM* **24**(5) pp. 300-309 (May 1981).

28. J. Reif and L. Valiant, "A logarithmic time sort for linear size networks," *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 10-16 (Apr. 1983).

29. W. J. Savitch, "Parallel random access machines with powerful instruction sets," *Mathematical Systems Theory* **15** pp. 191-210 (1982).

30. J. T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems* **2**(4) pp. 484-521 (Oct. 1980).

31. Y. Shiloach and U. Vishkin, "Finding the maximum, sorting and merging in a parallel computation model," *Journal of Algorithms* **2** pp. 88-102 (1981).

32. H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Transactions on Computers* **C-20**(2) pp. 153-161 (Feb. 1971).

33. E. Upfal, "Efficient schemes for parallel communication," *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, (1982).

34. E. Upfal, "A probabilistic relation between desireable and feasible models for parallel computation," Technical Report 83-18, Institute of Mathematics and Computer Science, The Hebrew University, Jerusalem, Israel (June 1983).

35. L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pp. 263-277 (1981).

36. L. G. Valiant, "A scheme for fast parallel communication," *SIAM Journal on Computing* **11** pp. 350-361 (1982).

37. U. Vishkin, "A parallel-design space distributed implementation space (PDDI) general purpose computer," Research Report RC9541, IBM Thomas Watson Research Centre, Yorkown Heights (1982).

38. U. Vishkin, "Implementation of simultaneous memory address accesses in models that forbid it," *Journal of Algorithms* **4**(1) pp. 45-50 (Mar. 1983).

39. A. Waksman, "A permutation network," *Journal of the ACM* **15**(1) pp. 159-163 (Jan. 1968).