AN INTENSIONAL PARALLEL PROCESSING LANGUAGE
FOR APPLICATIONS PROGRAMMING

by

E. A. Ashcroft, A. A. Faustini, R. Jagannathan

ARIZONA STATE UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Tempe, Arizona 85287-5406

TR-88-030
An Intensional Parallel Processing Language
for Applications Programming

by

E. A. Ashcroft
A.A. Faustini
Department of Computer Science
Arizona State University
Tempe
Arizona 85287

R. Jagannathan
Computer Science Laboratory
SRI International
Menlo Park
California 94025

Abstract

Parallel programming languages fall into two classes, the implicitly parallel languages and the explicitly parallel languages. We will consider the implicitly parallel languages.

The explicitly parallel processing languages are important in that they favor the implementation software. Compiler technology is relatively simple in this context. As a result we see a rapid proliferation of explicitly parallel languages on multi-processor architectures. This has important consequences for parallel processing in the short-term.

The implicitly parallel languages are equally important in that they favor the applications software. The software developed is more portable and more easily maintained over a software life cycle that can easily extend over successive generations of multi-processor architectures. This has important consequences for the long-term success of parallel processing.

Languages that are not based on a sequential model of computation can exhibit significant amounts of implicit parallelism. The declarative programming languages are therefore important parallel programming languages. Within the declarative paradigm we feel that the languages based on intensional logic are unique in that they permit data structures such as arrays, lists and tree to be implemented in a manner that is easily distributable.

To illustrate the concepts we have chosen a particular language namely, Lucid. However the same principles apply to other implicitly parallel languages. Lucid's simplicity and semantic elegance make possible the inclusion of a non-trivial application program - an adaptive solution to the n-body problem. We present an analysis of the implicit parallelism available in such a program and show ways in which we can harness this parallelism.

1. Introduction

Many scientific and engineering applications demand computing power and speed which cannot be provided by traditional supercomputers due to the inherent limitations of a single processor. Thus, much attention has been given to the possibility of distributing a given computation over an ensemble of cooperating processors. The programming languages used to code algorithms for such ensembles are parallel processing languages. These languages can be divided into two broad categories:
• explicitly parallel languages
• implicitly parallel languages

An explicitly parallel language is one in which the programmer has to explicitly state, using particular programming language constructs (fork, join, parbegin, etc.) or through program annotations, those parts of the program that can be executed in parallel. This has enormous advantages for the compiler technology used in implementing the language. It tells the compiler exactly where to find the parallelism in a program which in turn can lead to an efficient implementation. Moreover if the programmer knows how particular language features are implemented on a particular machine then even further performance advantages can be gained.

An implicitly parallel language is one in which the programmer has no say in the way the solution will be executed. The compiler technology has to completely determine a parallel execution strategy. The parallelism available in the program is implicit in the data dependencies of the program. These languages have great advantages for applications software. The programmer approaches solutions free from the details of a particular implementation and so is able to produce applications software that can be used over several generations of multi-processor architectures. Implementation abstraction also gives the compiler writer complete freedom to implement language features in a manner that cannot be interfered with from the user program.

We are thus faced with a dilemma. Do we want implementation-oriented software that runs well on today's multi-processors or do we want applications-oriented software that will last over generations of machines but for which there are few if any good parallel implementations as yet? Not much of a choice is it? Well, it depends on your needs. If you need an application to run in parallel today then you choose the first. If you are interested in challenging issues in compiling and programming languages you choose the second. We have chosen the second because we believe that in the long run it will be critical in determining the success of parallel processing.

2. Lucid, a Parallel Processing Language

In this section we will introduce the reader to the basic concepts of Lucid. The goal of the section is to provide the reader with enough understanding of the language to enable them to understand the application programs presented in the next section.

Lucid is a simple programming language in which programs are stated in terms of equations that express pure data dependencies. The simplest form of an equation is one that defines a variable to be equal to some constant (e.g. a=1). Sets of equations of this form are simple to understand but not very useful as a programming language. Lucid is a simple and elegant extension to this, based on Intensional Logic and the Lambda Calculus. Intensional Logic is a relatively modern branch of mathematical logic concerned with assertions and other expressions whose meaning depends on an implicit context. The relationship of intensional logic and programming languages is only now beginning to be understood [Faustini 87]. An intensional programming language is one based on a formal system based on intensional semantics. Intensional programmers are encouraged to think intensionally (when appropriate), and should be provided with context switching operators which allow values from different contexts to be combined without explicit context manipulation. Lucid is just such a language and the n-body problem described in the next section is an excellent example of an intensional program. Lucid incorporates the notion of a function as in functional programming [Peyton Jones 87] The result is a programming
language that expresses pure data dependencies naturally and has the potential for expressing algorithms that contain large amounts of implicit parallelism.

The description of an algorithm usually involves the concept of temporal order. For example, an algorithm might be defined in the following terms: "we first initialize the variables, then we compute the product of the variables after which we find a minimum and then ....". The notion of time expressed in this description is well known to programmers. It is not "real time" but relative time. Exactly the same concept exists in Lucid. To be more precise the Lucid programmer can define a variable that begins with an initial value, this is then followed by a second value and so on. This is in many ways a pure form of the assignment statement of imperative languages, the difference being that in Lucid the value that a variable has at a context can never change. The temporal context changes rather than the value of a memory location. Thus when the programmer revisits the same context the binding that was made in that context will still be valid. The following equation defines a variable that is \textit{first} 1.2 and then at successive time points 2.5, 3.8, 5.1 and so on. In other words the \textit{next} value in the temporal sequence is based on the previous value plus 1.2. In Lucid this is written as:

\[
x = 1.2 \ fby \ 1.3 + x
\]

We read this as \( x \) equals 1.2 followed by \( (fby) \) 1.3 plus \( x \). The preceding definition is an inductive definition but one can see that the data dependency is sequential. That is, the \( i \)th value in time depends directly on the \( i-1 \)th time. It is therefore not possible to exploit parallelism from this definition. On the other hand if a program consists of many such definitions it is possible to execute in parallel those equations that don't depend on each other. It is also possible to define a variable in terms of its own future as in the following example:

\[
x = \text{if} \ \text{now} < 1000 \ \text{then} \ \text{next} \ x \ \text{else} \ \text{now} \ \text{fi}; \]  
\[
\text{now} = 0 \ fby \ \text{now} + 1;
\]

Here we have two equations; one defines \( x \) in terms of its own future and the second is a counter. The reason that the counter is called \( \text{now} \) is that it corresponds to the temporal index of variables that use it. That is, at time 0 \( \text{now} \) is 0 and at time 1 it is 1 and so on. This is such a useful variable that it is included in the language as a built-in. This example illustrates another situation in which we have sequential temporal data dependencies. That is the value of \( x \) at time 0 is dependent upon \( \text{now} \) at time 1000, \( x \) at time 1 is dependent on \( \text{now} \) at time 1001 and so on.

These two example show us that simple temporal equations have a limited form of parallelism that comes mainly from the potential of non-interacting equations that can be executed independently.

How do we get more parallelism out of Lucid programs? We can extend the language from simple recursive equations to recursive function definitions. The following is an example of such a function that computes the prime numbers using the standard sieve algorithm:

\[
s\text{ieve}(n) = n \ fby \ \text{sieve}(n \ \text{whenever} \ n \ \text{mod} \ \text{first} \ n \ \text{ne} \ 0);
\]
\[
\text{primes} = \text{sieve}(2 \ fby \ \text{odds});
\]
\[
\text{odds} = 3 \ fby \ \text{odds} + 2;
\]

Here the parallelism associated with each recursive call can be exploited. Each call to \text{sieve} assumes that the first number, \text{first} \ n, it is sent will be a prime and it uses
that number to filter out (using the infix operator \texttt{whenever}) all subsequent inputs that are multiples of the first number. The resulting filtered temporal stream is then send on to another call of \texttt{sieve} that performs exactly the same task. Thus successive calls of \texttt{sieve} produce the primes in successive order. The parallelism comes from the fact that functions are always active as long as there is sufficient input in the pipeline (the pipeline being the temporal stream). It is well known that this type of parallelism can be exploited from recursive function definitions. The difference here is that we have the added benefit of pipelining through temporal streams. Recursive function equations (the Lambda Calculus languages) are not the only way of exploiting parallelism in declarative languages. In this paper we discuss a novel way of exploiting the parallelism associated with an intensional space.

Algorithms can involve spatial as well as temporal contexts. Examples of this are: the current employees of a company, the current temperature in different parts of the world, and so on. Lucid permits spatial as well as temporal contexts. The simplest form of spatial context, a vector, is very similar to the temporal stream. The vector of even numbers in increasing order is defined as follows in Lucid:

\[
\text{evens} = 0 \text{ sby evens+2;}
\]

Note we use \texttt{sby} (pronounced 'succeeded by') to generate successive values in space. The initial value of the vector is 0 then successive values are generated from the last value plus 2. We can obviously have multiple space dimension. In Lucid this is achieved by allowing the variables to range over an arbitrary number of space contexts. Thus if we want to generate the even numbers in the second space dimension we use the following equation:

\[
\text{evens} = 0 \text{ sby1 evens+2;}
\]

the \texttt{sby1} operator is analogous to \texttt{sby} but for the 2nd space dimension. similarly if we want to generate the even numbers in a third space dimension we write:

\[
\text{evens} = 0 \text{ sby2 evens+2;}
\]

So far each equation that we have defined varies over only one of the dimensions, in general a variable can vary in many dimensions. The following is a multi-dimensional version of the functional program \texttt{sieve} described above:

\[
\text{primes} = N \text{ sby primes wherever primes mod initial primes ne 0;}
\]

\[
N = 2 \text{ sby N+1;}
\]

In this program \texttt{primes} varies in both time and space. At time 0 \texttt{primes} is the vector of successive natural numbers beginning with 2. At time 1 it is the vector of successive natural numbers beginning with 3 but with all multiples of 2 removed, At time 2 the vector of successive natural numbers beginning with 5 but with all multiples of 2 and 3 removed and so on. Just as in the functional case given above each point in time has a sequential computation associated with it. That is the removal of the multiples of the prime found at that point in time. (In this case the Lucid built in filter \texttt{wherever}, pronounced 'where ever', is used to filter the spatial vectors associated with each time point of \texttt{primes}). We now begin to see other ways in which Lucid programs can be exploited for implicit parallelism. Figure 2.1 Shows diagrammatically the prime program just described. Each of the \texttt{wherever} operators can be working in parallel.
Figure 2.1 - Generating the primes using a multi-dimensional space

Lucid provides other context shifting operators. If we want to move forward in time we use the prefix operator next as in the example above. If we are in space we also have an analogous way of context shifting by using the prefix operator succ. (for successor). Thus

\[ x = \text{succ succ } y; \]

means that \( x \) at space point \( i \) is dependent upon \( y \) at space point \( i+2 \). Note that in all of these examples the time and space contexts are implicit or hidden. The operators succ and next allow a programmer to combine value for variables at different contexts. The two operators just described move the programmer forward in unit incremental steps. We also have operators that move the programmer backwards in unit increment steps. For time this is prev (for previous) and for space pred (for predecessor). The following equation illustrates the power of intensional programming as a notation for problem solving. It implements an iterative algorithm that is often found in numerical analysis and image processing applications. The idea is that we have a time varying matrix. At the first point in time we are given a matrix as input. At all times after that the value of the matrix at any point in space is dependent upon the value of its immediate neighbors at the previous point in time. In Lucid we write this as

\[ M = \text{given_matrix fby (succ M + pred M + succl M + predl M)}/4; \]

This illustrates the concept of parallelism through intensionality because it clearly shows that it is possible at successive points in time to compute in parallel all points of the new matrix.

The context shifting operations just described are very structured moves that led in general to programs that can be analyzed using formal techniques in order to prove properties about programs. Lucid also provides a potentially unstructured context shifting.
operation namely atspace and attime. These are potentially unstructured since the expression \( \mathbf{x} \text{ attime } \mathbf{N} \) means that we evaluate \( \mathbf{N} \) in the current context and use its value as the index of the next time point in \( \mathbf{x} \). On the other hand attime can be used in very useful and structured ways as in

\[
\text{pair\_sum} = (\mathbf{v} + \text{succ } \mathbf{v}) \text{ atspace } (2 \times \text{here});
\]

In this equation here is the space analog of now, that is, at space point 0 here is 0, at space point 1 it is 1 and so on. The above program defines a vector pair_sum that is the result of consecutive pairwise summation of \( \mathbf{v} \) in space. That is \((v^0+v^1), (v^2+v^3), (v^4+v^5)\) and so on. Note that this is not achieved by either of the following equations:

\[
\begin{align*}
\text{shift1\_sum} &= \mathbf{v} + \text{succ } \mathbf{v}; \\
\text{shift2\_sum} &= \mathbf{v} + \text{succ succ } \mathbf{v};
\end{align*}
\]

the first of which gives us a pairwise summing of \( \mathbf{v} \) that corresponds to \((v^0+v^1), (v^1+v^2), (v^2+v^3)\),\ldots\; and the second of which gives us a pairwise summing that corresponds to \((v^0+v^2), (v^1+v^3), (v^2+v^4)\),\ldots\;Note that in all of these examples addition can proceed in parallel provided that the vector \( \mathbf{v} \) has its values ready to use. We can easily extend the pair_sum definition to produce a parallel binary tree addition as illustrated in Figure 2.2.

The program of figure 2.2 include two concepts that have not yet been explained. The first is that of a Lucid where-clause (or block). There are two types of Lucid where-clause. The first is a simple where-clause which is used to introduce auxiliary definitions of variables that can be used in the block itself or in the head of the where-clause. The following example illustrates the simple where-clause:

\[
\begin{align*}
\text{x where } & \text{x = y; } \\
& \text{y = z; } \\
& \text{z = 1 fby z+1; } \\
\text{end;}
\end{align*}
\]

The \( \text{x} \) before the word \text{where} is the head (or subject) of the where-clause. It directly corresponds to the value of the whole construct. The definitions within the where-clause are simply auxiliary definitions. The where-clauses can be nested and when they are they have almost the same scope rules as a Pascal block. The only difference is that the head of the clause which appears outside the where-end construct is also in the scope of the block.

The second type of where-clause in Lucid is one which contains declarations of variables using \text{is current} or \text{is s_current}. These blocks conform to the same scope rules a simple where-clauses but the have an added duty. They freeze the value of the variables declared at the beginning of the where-clause. This has the effect of a subcomputation. In the program in figure 2.2 the values of the formal parameters to the function binary_sum are both frozen in time. This means that when the function is executed the time associated with the call of binary_sum is frozen this enables the subprogram to use its own local time to process the vector \( \mathbf{v} \) and the (now constant) \( n \). \( 2^n \) values of \( \mathbf{v} \) are added). When a value is returned for the sum of vector \( \mathbf{v} \) it is returned to the point in time at which the call was made. Thus the values of the actual parameters to binary_sum can be changing with time on the outside of the function but when they are passed into the function that outside time is stopped and the local (inner) time takes over.
This construct is probably the most difficult to explain but it is very useful in intensional programs particularly in the design of library functions.

\[
\text{binary\_sum}(v,n) = \text{tree attime } N \\
\quad \text{where} \\
\quad \quad v \text{ is current } v; \\
\quad \quad N \text{ is current } n; \\
\quad \quad \text{tree}=\text{v fby (tree+succ tree) atspace (2*nhere);} \\
\quad \text{end;}
\]

Figure 2.2 Parallel binary addition of a vector

Finally, for those readers who prefer a more formal description of some of the operators, we have provided the following definitions. The following operators are defined over a flat intensional space consisting of a single time context and an arbitrary number of spatial contexts:

Data and conditional computations do not change context when evaluating their arguments. We call these operations pointwise since there evaluation does not change the current context.
\[
(\text{if } <E1> \text{ then } <E2> \text{ else } <E3> \text{ fi})_t^{s_0, s_1, s_2, \ldots} = \begin{cases} 
(\text{if } <E1>_t^{s_0, s_1, s_2, \ldots} = \text{true then } <E2>_t^{s_0, s_1, s_2, \ldots} \\
(\text{if } <E1>_t^{s_0, s_1, s_2, \ldots} = \text{false then } <E3>_t^{s_0, s_1, s_2, \ldots}
\end{cases}
\]

\[
(<E1> \text{ } \text{pointwise op} \text{ } <E2>)_t^{s_0, s_1, s_2, \ldots} = (<E1>_t^{s_0, s_1, s_2, \ldots} \text{ pointwise op } <E2>_t^{s_0, s_1, s_2, \ldots})
\]

The following are some of the temporal operators:

Let \( t, s_0, s_1, s_2, \ldots \) range over the integers.

\[
(<E1> \text{ fby } <E2>)_t^{s_0, s_1, s_2, \ldots} = \begin{cases} 
\text{if } t = 0 : (<E1>_0^{s_0, s_1, s_2, \ldots} \\
\text{if } t > 0 : (<E2>_{t-1}^{s_0, s_1, s_2, \ldots}
\end{cases}
\]

\((\text{now })_t^{s_0, s_1, s_2, \ldots} = (t)\)

\((\text{next } <E1>)_t^{s_0, s_1, s_2, \ldots} = (<E1>_{t+1}^{s_0, s_1, s_2, \ldots})\)

\((<E1> \text{ attime } <E2>)_t^{s_0, s_1, s_2, \ldots} = (<E1>_{<E2>}^{s_0, s_1, s_2, \ldots})\)

first <E1> is <E1> attime 0

Finally the remaining equations describe some of the context shifting operations for Lucid's space dimensions:

Let \( t, s_0, s_1, s_2, \ldots \) range over the integers.

\[
(<E1> \text{ atspacek } <E2>)_t^{s_0, s_1, s_2, \ldots} = (<E1>_t^{s_0, s_1, s_2, \ldots}) \cdot (<E2>_{s_{t+1}}^{s_0, s_1, s_2, \ldots}, s_{t+1}, s_{t+1}, \ldots)
\]

\[
(<E1> \text{ sbyk } <E2>)_t^{s_0, s_1, s_2, \ldots} = \begin{cases} 
\text{if } s_k = 0 : (<E1>_t^{s_0, s_1, s_2, \ldots}) \\
\text{if } s_k > 0 : (<E2>_t^{s_0, s_1, s_2, \ldots})
\end{cases}
\]

\((\text{herek })_t^{s_0, s_1, s_2, \ldots} = (s_k)\)

\((\text{succk } <E1>)_t^{s_0, s_1, s_2, \ldots} = (<E1>_t^{s_0, s_1, s_2, \ldots, s_{t+1}, s_{t+1}, \ldots})\)

initialk <E1> is <E1> atspacek 0
3. An Application in Lucid: The n-Body Problem

The main purpose of this section is to show that it is possible to program a real application in Lucid, and exhibit the implicit parallelism in it. In the process we will show, we hope, that Lucid programming is different but not incomprehensible.

The main application of Lucid that we will consider is the so-called n-body problem. It is concerned with simulating the movement of a set of bodies in space, under the influence of, in this case, gravity. (The same program, with minor modifications, will work equally well for electromagnetism.) The crucial characteristic of this problem is that forces between bodies obey an inverse-square law. This means that effects between bodies are felt at large distances, and every one of them bodies feels the effect of the other n-1 bodies.

The traditional technique for solving this problem involves setting a time-increment "delta", and having all the bodies' positions and velocities calculated at particular times, the successive times being separated by intervals of size delta. These successive periods at which calculations are performed are called "stages", and in this technique every body goes through the same stages. This solution is relatively easy to program in Lucid. Here is the program:

```lucid
[% p, v %] wherever here < n
where
  p = p0 fby p + v*delta + (a*delta*delta)/2
  where p0 = p0x sby1 p0y sby1 p0z; end;
  v = v0 fby v + a*delta
  where v0 = v0x sby1 v0y sby1 v0z; end;
  a = initial binary_sum(accel_contrib,power_of_two)
  where
    BODY is s_current here;
    p is s_current p;
    int_p = p;
    accel_contrib = if here eq BODY or here>=n then 0
                   else G*mass/norm_sq(dist)*projection fi;
    dist = int_p - p;
    projection = dist/norm(dist);
  end;
  sum(a) = initial1(a + suc1(a) + succl succ1(a));
  power_of_two = now asa 2**now>=n;
  norm_sq(a) = sum(a*a);
  norm(a) = sqrt(norm_sq(a));
  binary_sum(h,n) = b asa atime READY
                     where
                       H is current h;
                       READY is current n;
                       b = H fby (b+suc b) atspace (2*here);
  end;
```

The variables p, v, and a are defined to be the positions, velocities and accelerations, respectively, of the various bodies. (The program outputs the positions and velocities of the bodies at successive stages in time.) The meaning of any variable, p for example, depends
on the context in which it is desired. That context will say which body is being considered and which stage in the trajectory of that body is being considered, and even which dimension or direction is being considered, x, y, or z. The crucial, and characteristic, feature of Lucid is that the context will be implicit. Contexts are not mentioned explicitly in a Lucid program; the definitions are for all contexts. The one definition for p, for example, covers each body at each stage in each direction. Lucid programs are intentional. In the following discussion it might help if we assume that we are considering some generic implicit context, or intension, C.

The definitions for p and v are relatively straightforward. They are based on well-known equations of physics. Also, they use the Lucid operator $\equiv$ by ("followed by"), but are easily understood. The definition of p, for example, can be understood as saying that p starts off (at the initial stage) as being p0 (which has x, y, and z components, p0x, p0y, and p0z ($\equiv$ by means "succeeded by"), and then, at any stage, the value of p at the next stage is the value of $p + v \cdot \text{delta} + (a \cdot \text{delta} \cdot \text{delta}) / 2$ at the current stage. Equivalently, the definition of p says that if the current implicit context C says that we are at the initial stage then the value of p in context C is the initial value p0 in the appropriate direction (that direction being specified by C). On the other hand, if C says we are at a later stage, the value of p is the value of $p + v \cdot \text{delta} + (a \cdot \text{delta} \cdot \text{delta}) / 2$ in the context corresponding to the previous stage (but to the same body and direction).

**Figure 3.1 - Version 1 (4-Bodies with fixed delta)**

Note: Initial velocities and masses of the Bodies are the same for all Figures

The definition for a is more complicated than that for p. To determine the (gravitational) acceleration in context C, one has to use a subcomputation to sum the contributions of all the other bodies, using the masses of the other bodies and the distances (dist) to them. The function binary_sum, in context C, adds together the values of its argument, accel_contrib, for all similar contexts, that differ from C only in the body
being considered. (That is, if a context $B$ is similar to $C$, it is for the same stage, and for the same direction, $x$, $y$, or $z$, but for a different body.) By suitably defining $\texttt{accel\_contrib}, \texttt{binary\_sum} (\texttt{accel\_contrib})$ adds together the acceleration contributions (in the $C$-direction) of all the bodies at the same stage (as specified by $C$). For each of the $C$-similar contexts $B$, the value of $\texttt{accel\_contrib}$ depends on mass, the mass of the $B$-body (the body for the context $B$), and on $\texttt{dist}$, the distance from the $B$-body to the original body we were considering (the $C$-body). To do that, when we switch to the new context $D$, we need to remember the position of the $C$-body. That is done by the declaration $\texttt{p}$ is $\texttt{s\_current\_p}$. Its effect is to cause $\texttt{p}$ to have the $C$-value of $\texttt{p}$ in all $C$-similar contexts. The distance $\texttt{dist}$ is then simply the difference between the position $\texttt{p}$ and the position of the $C$-body, namely $\texttt{p}$. (We define $\texttt{dist}$ as $\texttt{int\_p - p}$, but we said elsewhere in the program that $\texttt{int\_p is p}$, so $\texttt{dist}$ is defined correctly. The reason for introducing the superfluous variable $\texttt{int\_p}$ will become clear when we describe a later version of the program.)

The rest of the definition of $a$ should be understandable. In the denominator of the expression for $\texttt{accel\_contrib}$, we use $\texttt{norm\_sq(dist)}$, rather than just $\texttt{dist*dist}$, because the $x$ component, say, of the force on a body depends on the magnitude of the distance, not just the distance in the $x$ direction (which could be very small, even though the bodies are far apart). ("Norm" is the term used in vector algebra for the magnitude of a vector.) To get the $x$, $y$, and $z$ components of the acceleration contribution, we have to project, into the required directions, the value obtained from the magnitude of the distance (hence the use of projection).

The program is quite short, but it is important to appreciate that it handles any number of bodies without ever referring to any of them directly, just by using generic-looking definitions. It "outputs" the positions and velocities of the various bodies at each stage. It will do so by first asking for $n$, the number of bodies, and then asking for the initial positions and velocities (in each of the $x$, $y$, and $z$ directions) of the various bodies. (The bodies will be numbered 0 through $n$-1.) The initial (stage 0) positions and velocities will then be printed out. It will then ask for the masses for bodies 1 through $n$-1, and output the position and velocity of body 0 at stage 1. It will then ask for the mass of body 0, and output all the positions and velocities of all the masses at all stages. (The output can be terminated with control-D, or the program could easily be modified to output a predetermined number of stages.) The mass of body 0 is asked for last because it is not needed until the acceleration, at stage 0, of body 1 is needed. (The acceleration, at stage 0, of body 0 is determined by the masses of bodies 1 through $n$-1.)

Figure 3.2 shows, in graphical form, the positions calculated by this program for four bodies started with different positions and velocities. For simplicity, in order to get output in two dimensions, the initial values in the $z$ direction were all given as 0. As well as the number of bodies, their masses, and their initial positions and velocities, the program requires the value of $g$, the universal gravitational constant, and $\Delta t$, the time increment. These two values could have been built into the program, but, as written here, their values will be requested when needed in a particular context. When needed in a different context than that one, a new request will be issued. This gets tedious for $g$, because it really is a constant and should have been built into the program, but $\Delta t$ might be different for different situations, and should be a parameter of the program. In fact, for an accurate simulation, the time interval $\Delta t$ has to be chosen to be small enough to ensure that the acceleration is practically constant on any body during $\Delta t$, throughout the whole trajectory. This can be tricky. There may be situations that occur during a trajectory that call for a very small $\Delta t$, such as when one body is rotating about another, and it may not be apparent from the initial situation that such a small $\Delta t$ will be necessary. The time during a trajectory when such a small $\Delta t$ is needed may be
quite short, but the small \texttt{delta} has to be used throughout, requiring a large amount of computation that is perhaps unnecessary.

\textbf{Figure 3.2 - Version 2 (4 bodies global variable delta)}

All this suggests that it may be advantageous to have a \texttt{delta} that \textit{varies}. In fact, in Lucid it is more "normal" to have varying things than constant things. It is very easy to allow \texttt{delta} to vary - the code already given does not require \texttt{delta} to be constant! But how should it vary? The simplest way, requiring the least change in the code, is to have \texttt{delta} varying at each stage, but being the same for all the bodies, at each stage. The advantage of having \texttt{delta} the same for all bodies at each stage is that at each stage every body will be at the same real time. \textit{As a result, the Lucid program already given can be used; it need not be changed at all, except for the addition of a definition for \texttt{delta}!}

The only remaining question is the following: in what way is \texttt{delta} to vary? What we will do is calculate at each stage a candidate \texttt{delta} for each individual body, and then take \texttt{delta} as the minimum of the candidates. But how do we find the candidate \texttt{delta}'s? This question will temporarily lead us into discussions of low-level physics.

As we have said, for the method of calculation to be accurate, \texttt{delta} should be short enough for the acceleration of any body to remain essentially constant throughout \texttt{delta}. The acceleration will change if the distance moved in time \texttt{delta} alters the force on a body appreciably, causing the body to experience a different acceleration. This can happen in two ways.

The acceleration might be perpendicular to the velocity, causing the body to follow a curved trajectory, and cause a change in the \textit{direction} of the acceleration. Or, on the other hand, the acceleration may be in the same direction as (or opposite to) the velocity, causing the body to move closer to (or farther away from) the cause of the acceleration, thereby
changing the force on the body, and the *magnitude* of the acceleration. (In most situations, the acceleration and velocity are at an angle that isn't a right angle, and both factors have to be taken into consideration, and the smaller candidate chosen.)

In the first case, the ratio of the velocity and the acceleration perpendicular to it is proportional to the time taken to make a turn of a given angle. In other words, if we want to limit the angular turn to a certain amount, this ratio should be useful in limiting the time allowed. Some coefficient times the ratio is a good candidate for \( \delta \) in this case.

In the second case, there doesn't appear to be any simple combination of acceleration and velocity that can limit the time to allow for the acceleration to change by a certain percentage. The acceleration is caused by some other mass at some distance away, and using just the acceleration and the velocity we can not separate out these two factors in the cause, mass and distance. We would *need* to separate them out, because they have wildly different effects. If the body moves a given distance closer to the other mass in a given time, the acceleration will change more markedly if the other mass is small but very close than if it is large but far away. Therefore, to specify \( \delta \) in this case we shall resort to predicting \( \delta \) from the way that the acceleration has been varying in the past. We will need to "jump start" this process by specifying some initial value for \( \delta \).

The second version of the n-body program will be identical to the first except it contains, in the outer where clause, the following definition of \( \delta \):

```plaintext
\delta = \text{initial binary_min(candidate, power_of_two)};
candidate = \text{if here} \geq n \text{ then initial candidate else min(method1, method2) fi; }
method1 = C*(vv/aa)/\sqrt{1-\sqrt{\text{sum(a*vv)/(aa*vv)}}};
method2 = \text{jump_start sby D*delta/abs((next aa)/aa-1)};
vv = \text{norm(v)};
aa = \text{norm(a)};
\text{binary_min(h, n) = b at time N where}
  H \text{ is current n; }
  N \text{ is current n; }
  b = H fby \text{min(b, succ b) at space (2*here)};
end;
```

The definition of \( \text{candidate} \) uses two dimensionless coefficients, one, \( C \), corresponding to the first method of determining \( \delta \) as explained above, and the other, \( D \), for the second method. The values of \( C \) and \( D \) can be obtained by experiment. Smaller and smaller values should be tried until the trajectories of the various bodies stabilize.

In Figure 3.2 we show the trajectories obtained with Version 2 for exactly the same masses and velocities as for Figure 3.1. Clearly, the trajectories are rather different. For a fair comparison, the \( \delta \) for Version 1 should be set to the smallest \( \delta \) obtained in Version 2. When that is done, the trajectories are very similar. The running time will be significantly different, however, because, in running Version 2, \( \delta \) varies by a factor of about 1000! Using the smallest \( \delta \) of Version 2 in Version 1 increases the running time by at least 100 times.
An interesting modification of Version 2 is obtained when we let \( \text{delta} \) vary from body to body within stages, as well as between stages. This will imply that the bodies have their own \textit{real times}, and this causes a problem if we try to use the solution described before. In Version 1, the definition of \( \text{dist} \) as \( \text{int}_p - \text{p} \) is critical to the definition of \( \text{accel_contrib} \). (We will be referring to notation used earlier when we were explaining Version 1.) This gives the distance from the \( B \)-body to the \( C \)-body. The context \( B \) is \( C \)-similar, so it has the same stage. In the first two versions of the program, "the same stage" implies "the same real time", and it is "the same real time" that is really essential. If we allow each body to have its own real time, and define \( \text{int}_p \) as \( p \), as before, we will be calculating the distances between objects at different real times (if the program calculates anything at all!), and the simulation falls apart. What we want to do is define \( \text{int}_p \) to be such that in context \( B \) it gives us the position of the same \( B \)-body, but at the same real time as the \( C \)-body had. (The name \( \text{int}_p \) is short for "interpolated position". The desired position will probably fall between positions that are calculated as values of \( p \), and we must interpolate.)

![Figure 3.3 - Version 3 (individual deltas)](image)

This is done by finding the stages in the trajectory of the other body that correspond to the real times of all the stages for the \( C \)-body, as closely as possible. (That is, the real times that get as close as possible to the real times for the \( C \)-body, without going past them.) The differences, \( \text{diff} \), between these real times and the \( C \)-body real times then are then used to calculate the interpolated positions, based on the positions, velocities and accelerations of the other body at these stages. All this can be seen in the definition of \( \text{int}_p \), \( \text{stage} \), and \( \text{diff} \) in Version 3.
The only other thing needed in Version 3 is a definition for all the real times. This is
the definition of $\tau$, as shown here in the outermost where clause of Version 3 of the
program.

[$% p \quad v \%$] wherever here $< n$
where
\[
\begin{align*}
\tau & = 0 \text{ sby } \tau + \delta; \\
p & = p_0 \text{ fby } p + v \cdot \delta + (a \cdot \delta \cdot \delta)/2 \\
v & = v_0 \text{ fby } v + a \cdot \delta \\
a & = \text{binary_sum}(\text{accel_contribution, power_of_two})
\end{align*}
\]
where
\[
\begin{align*}
\text{BODY} & \quad \text{is s_current now;} \\
\text{RT} & \quad \text{is current } \tau; \\
P & \quad \text{is s_current } p; \\
\text{int}_p & \quad \text{p atspace stage +} \\
& \quad \text{if diff eq 0 then 0} \\
& \quad \text{else v atspace stage*diff+} \\
& \quad \text{(a atspace stage*diff*diff)/2} \\
fi; \\
\text{stage} & \quad \text{now asa rt eq c_RT or rt<c_RT and next rt>c_RT} \\
& \quad \text{where c_RT is current RT; end;} \\
\text{diff} & \quad \text{= RT - rt atspace stage;} \\
\text{accel_contribution} & \quad \text{if BODY eq now then 0} \\
& \quad \text{else G*mass/norm_sq(dist)*} \\
& \quad \text{projection} \\
fi; \\
\text{dist} & \quad \text{= int}_p - P; \\
\text{projection} & \quad \text{= dist/sqrt(norm_sq(dist));} \\
end; \\
\text{delta} & \quad \text{= initial binary_min(candidate, power_of_two);} \\
\text{candidate} & \quad \text{if here}==n \text{ then initial candidate} \\
& \quad \text{else min(method1, method2)} \\
fi; \\
\text{method1} & \quad \text{= C*(v*v)/aa/sqrt(1-sqrt(sum(a*v)/(aa*v*v)))}; \\
\text{method2} & \quad \text{= jump_start sby D*delta/abs((next aa)/aa-1)}; \\
v & \quad \text{= norm(v);} \\
a & \quad \text{= norm(a);} \\
\text{binary_min}(h, n) & \quad \text{= b attime } N \\
\text{where} \\
& \quad H \text{ is current h;} \\
& \quad N \text{ is current n;} \\
& \quad b = H \text{ fby min(b, succ b) atspace (2*here);} \\
end; \\
\text{sum}(x) & \quad \text{= initial1(x + succl x + succl succl x);} \\
\text{norm}(x) & \quad \text{= sqrt(norm_sq(x));} \\
\text{norm_sq}(x) & \quad \text{= sum(sqr(x));} \\
\text{binary_sum}(h, n) & \quad \text{= b attime } N \\
\text{where} \\
& \quad H \text{ is current h;} \\
& \quad N \text{ is current n;} \\
& \quad b = H \text{ fby (b + succ b) atspace (2*here);} \\
end;
In figure 3.3 we show the trajectories obtained with version 3. Note that we let the trajectories run on much longer to show some interesting behavior of bodies 3 and 4. The close up of the area around position (0,0) is shown in figure 3.4. This probably does not represent the last version of the program, but it is anticipated that any further refinements, or modifications, can be made in just as unintrusive a manner as the preceding two. One of the reasons for going through the development of this program in such detail has been to show how straightforward it is to modify Lucid code, instituting radically new ways of computing the desired result by making local changes.

4. Efficient Execution of Lucid Programs

A Lucid program expresses parallelism implicitly. This property of the language is of significance if the inherent parallelism in a Lucid program can be effectively and automatically exploited. Effective exploitation of parallelism means maximal exploitation of parallelism in useful computing. Automatic exploitation of parallelism means using a general computing model to exploit inherent parallelism from any Lucid program without user assistance.

First, we briefly discuss why traditional control-driven computing models are inappropriate for evaluating Lucid programs. Instead, we describe a general computing model called eduction that allows for effective and automatic exploitation of parallelism inherent in a given Lucid program. Then, we describe an abstract architecture that embodies the eduction computing model and consider two implementations of the abstract architecture: a single board computer and a parallel computer.

Control-driven Computing Models

Control-driven computing models (which are also known as von Neumann computing models) assume that a computation is represented procedurally as a sequence of
operations to be applied on data which is at rest. The declarative nature of a Lucid program does not naturally suggest such a procedural representation. Instead, a Lucid program can be best viewed as a network of operators applied to data in motion which is contrary to the procedural view [Wadge 85]. Therefore, we conclude that control-driven models are inappropriate for evaluating Lucid programs. Furthermore, computing models that assume a network of operators as representation of computation are appropriate for evaluating Lucid programs. One such computing model is eduction.

Eduction Computing Model

In operational terms, each variable or term of a Lucid program denotes an unordered set of value-holders each of which is identified by its relative conceptual position amongst all value holders for that variable or term. We use the word daton to refer to a particular value-holder of a variable or term. For example \( v_2 \) denotes the \( 2^{\text{nd}} \) daton of the variable or term denoted by \( v \). Note that \( q \) is the context or intension which we refer to as the tag of the daton. Each operation of a Lucid program denotes a function which computes the values for the datons of the defining term or variable with values of datons of its argument terms and variables. We think of a computation to be the execution of a Lucid program given a set of sequences of input daton values (over time) and a sequence (over time) of needs for values of output datons of the program.

A computing model conducts the computation in steps. A computational step is a logical concept and does not necessarily correspond to physical time. At the initial step, none of the values of datons except possibly for the values of input datons are defined. At each subsequent step, some of the undefined datons are computed. Eventually, the values of needed output datons are computed.

We claim that a computing model can be described using the following two characteristics.

1. Which daton values are to be computed, and
2. When these daton values are to be computed.

Independent of the computing model, the sequence of needs for output daton values, in fact, indicates which output daton values are needed, and if an output daton value is needed, at which stage it is needed.

It is important to note that all computing models implement the denotational semantics of each operation faithfully. The operational function applied to argument daton values to produce a result daton value is the same for all computing models. Furthermore, this operational function is the correct interpretation of its mathematical counterpart.

We require the computing model that we use to execute Lucid programs to be efficient. In defining efficiency, we assume that the underlying implementation has a finite amount of processing, storage, and communication resources using which the computation needs to be conducted. An efficient computing model should exploit as much useful parallelism as possible while bounding the amount of useless parallelism exploited. We say that a daton is useful if its value determines value of some needed output daton; otherwise, the daton is useless. Restated, an efficient model should compute as many useful daton values as possible, as soon as possible, while computing only a bounded number of useless daton values.
Consider a simple computing model which requires all daton values to be computed as soon as possible. We call this the *eager* computing model. While the eager model exploits as much parallelism as possible, it incurs an unbounded amount of waste in the form of computed daton values that are useless. Such a model is therefore inefficient. Data-driven dataflow architectures [Treleaven 82, Jagannathan 88] are implementations of variants of the eager computing model. Inefficiencies manifest in these architectures as resource bottlenecks.

Now consider the eduction computing model [Ashcroft 86]. The word *education* is defined in the Oxford English Dictionary as follows: 'The action of drawing forth, eliciting, or developing from a state of latent, rudimentary, or potential existence; the action of educating (principles, results of calculations) from the data.' The eduction computing model can be concretely described using the notion of demands. In particular, a daton value is demanded when it is required to be computed and the demand for the daton value is satisfied when the daton value is defined. Note that demand for value of a constant daton or an available input daton is satisfied immediately.

The eduction computing model can be succinctly described as follows:

(a) Need for an output daton value at some step causes the daton value to be demanded at that step.
(b) If a daton value \( a_i \) is demanded at some step, then and only then are values of all datons that are known to determine value of daton \( a_i \) demanded.

Output daton values are demanded at the steps at which they are needed. A demand for a daton value causes demands for those daton values that are known to determine it. For example, consider the following definition:

\[
x = \begin{cases} 
  a & \text{if } p \\
  b & \text{else}
\end{cases} \ast y
\]

Demand for value of \( x_i \) at step \( t \) causes demands for values of \( p_i \) and \( y_i \) at step \( t + 1 \) because values of only these datons are known to determine value of daton \( x_i \) at step \( t \).

Assuming that the value of \( p_i \) is defined at step \( s \) and it is \( T \), the daton value \( a_i \) can be demanded at step \( s + 1 \) since this value is now known to determine value of \( x_i \).

We claim that the eduction computing model does not demand (and therefore compute) values of any useless daton. (A formal proof is beyond the scope of this paper. Interested readers are referred to [Jagannathan 88].) The model satisfies the condition for efficiency which is that the amount of useless parallelism exploited should be bounded (which in the case of the eduction model is zero). However, in general, the amount of useful parallelism exploited when using the eduction model is less than optimal. This is because the model demands values of datons only when they are known to be useful in its effort to bound useless computing. For example, in the above definition of variable \( x \), demand for value of \( x_i \) immediately only results in demand for value of \( p_i \) and \( y_i \) since the utility of \( a_i \) or \( b_i \) depends on the value of \( p_i \). Furthermore, one of these values is demanded only when the value of \( p_i \) is available. If avoiding useless computing were not a concern, demand for value of \( x_i \) would immediately cause values of \( p_i, a_i, b_i \), and \( y_i \) to be demanded resulting in greater parallelism.
It is reasonable to assume that, in an implementation of the eduction computing model, the cost of demand propagation as required by the model and the cost of executing tag-manipulation operators (such as $\text{copy}$, $\text{at\space space}$, $\text{succ}$) is significantly lower than the cost of executing arithmetic and logic operations such as addition, multiplication, comparison. Therefore, the apparent slowdown due to demand-propagation is usually insignificant.

We illustrate how the eduction computing model effectively exploits useful parallelism inherent in the $\text{binary\_sum}$ function. Assume that the time taken to process and propagate a demand and the time taken to execute tag-manipulation operators of Lucid are significantly smaller than the time taken to execute (a possibly floating-point) operation such as an arithmetic or logic operation. Consider the $\text{binary\_sum}$ function defined in the $\text{nbody}$ program. The $\text{binary\_sum}$ function theoretically computes the sum of a (spatial) sequence of numbers of length $n$ in $O(\text{ceiling}(\log_2 n))$ time which is optimal.

Processing of demand for $\text{binary\_sum}(h, n)$ at Lucid time $t$ and space $0$ is shown in figure 1.

![Diagram](image)

Figure 4.1: Propagation of demands to compute $\text{binary\_sum}(h,n)$

Initially, demand for sum of all numbers propagates as demands for the sum of each half of the sequence of numbers which, eventually, causes demands for each of the numbers. Each pair of numbers is added together to produce a sequence of sum of pairs of numbers. This
is repeated again producing a sequence of sums of quadruplets of numbers. Eventually, the sum of all numbers is computed. This is illustrated in figure 4.2.

\[ ? \text{binary_sum}(h, n) @ (t, 0) \]

Figure 4.2: Computation tree for binary_sum( h, n )

Given that there are \( n \) numbers to add such that \( m \) is the smallest power of 2 no less than \( n \), the total number of additions when using the eduction model is

\[ \frac{m}{2} + \frac{m}{4} + \ldots + 1 = m - 1 \]

These are performed by the eduction model in \( \log_2 m \) steps for an average parallelism of \( O(m \log_2 m) \). If the eager model we referred to earlier were used, the total number of
additions would have been $2m \cdot \log_2 m - 1$ which is almost twice the number of additions when using the eduction model. Since $n - 1$ is the lower bound on the number of additions to compute the sum of $n$ numbers, the eduction model exploits useful parallelism in near-optimal time (ignoring demand propagation and tag manipulation times which we assume are nominal). The eager model does almost as many superfluous additions as useful additions while not spending any time on demand propagation.

It is possible to improve the speed of the eduction computing model by computing values of not-yet-useful datons anticipatorily. The resulting computing model is called eazyflow [Jagannathan 88]. In the eazyflow computing model, for certain variables (called "eager"), demanding a value of a daton known to be useful causes a bounded number of not-yet-useful datons (for the same variable) to also be demanded. This way when these values are demanded they will be available sooner. Consider the following program to add the first $2^n$ power roots of a number using binary_sum

```c
binary_sum( x, n ) where
  x = 0 cby sqrt(x);
end
```

Assuming variable $x$ is eager, when a value of $x_i$ is demanded by fastsum, a few extra daton values (of $x_{(i+1)}$ through $x_{(i+b)}$) are demanded at the same time. So, when values of $x_{(i+1)}$ through $x_{(i+b)}$ are demanded subsequently by binary_sum, they are already (being) computed because of the anticipatory demands for them. The eazyflow computing model can be shown to exploit at least as much useful parallelism as the eduction computing model while exploiting only a bounded amount of useless parallelism [Jagannathan 88].

Now consider the execution of the nbody program discussed in the previous section using the eduction computing model. Demanding $p$ and $v$ at some Lucid time and position $i$ for all $i$ less than $n$ corresponds to demanding the position and velocity of each of the $n$ bodies at that Lucid time. This causes the positions and velocities of bodies to be computed according to the definitions for $p$ and $v$. In doing so, the acceleration of each body is computed using the definition of $a$ which is the (binary) sum of the contributions of the other bodies. In steady state, at some Lucid time, demanding the position and velocity for all bodies causes the accelerations of all bodies to be demanded and computed in parallel, and the positions and velocities of all bodies to then be computed in parallel. This results in fairly effective exploitation of useful parallelism in the nbody program.

When using the eazyflow computing model, assuming variable $p$ and $v$ are eager in the time dimension, demanding the position and velocity of all bodies at a given time, causes the position and velocity of the bodies at subsequent times to be demanded anticipatorily. This means that when the positions and velocities of bodies are actually demanded at subsequent and possibly non-contiguous times, they would be computed much sooner than with the eduction computing model. In other words, the eazyflow computing model exploits the useful parallelism better than the eduction computing model while incurring some bounded amount of waste.

An Abstract Architecture

We are interested in an abstract architecture that embodies the eduction computing model. Such an abstract architecture is shown in Figure 4.3. The architecture is intended to suggest the design of an interpreter (in hardware or software). In the next section, we
propose two implementations of the abstract architecture, one of which is a pipelined implementation and the other is a parallel implementation. The abstract architecture (see Figure 4.3) consists of four components: processor pool (P), interface (I), value store (V), and term store (T). The processor pool consists of a number of processing elements that are capable of executing all the basic arithmetic and logical operators of Lucid. The interface allows for the evaluator to communicate with the external world. The value store retains names of datons, their values, and status of values along with a list of demanders. The term store consists of the program being executed and a list of suspended operations waiting for values that have been demanded.

![Diagram of Abstract Architecture]

**Figure 4.3: Abstract Architecture**

Program execution begins by a demand for value of an output daton at the switch. The demand is processed at the value store to determine if it has already been computed. If so the value is returned to the demander, which in this case is the external world. If not, the daton name is recorded in the value store with its status as "being computed" and the demand is sent to the term store.
The appropriate operation template is created and each value that the operation needs to execute is demanded. The demands bypass the processor pool and the interface and go directly to the value store.

If the value has not been computed yet or is being computed, the demand is retained in the store so that it can be satisfied when the value is eventually computed. If the value already exists, it is returned to the term store where it is inserted into the appropriate suspended-operation template.

If the arrival of a value at the term store causes a suspended operation to be executable, it is sent to the processor pool where the operation is applied and the result is sent via the switch to the value store or to the term store.

Value arriving at a value store causes the value to be recorded for the appropriate daton, its status to be changed to "computed" and all pending demands for the value to be satisfied and the values to be sent to the term store.

Implementations of the Abstract Architecture

We describe two computers that are based on the abstract architecture presented above. The first is a single-board pipelined implementation that can be used with compatible existing computers. The second is a multi-board parallel implementation that is scalable in performance.

A Single-Board Pipelined Computer

The abstract architecture proposed above can be directly implemented using conventional technology. The result is a single-board computer which essentially has four components: the term store, the value store, the interface, and the processor pool. Each of these components is connected to a common bus such that any two components can directly communicate with each other without requiring active participation from the other components.

The term store consists of a conventional processor with a large memory. The processor efficiently executes all Lucid tag-manipulation operators. It is responsible for demand-propagation and creation and management of suspended operations.

The value store consists of a conventional processor with a large memory. It implements associative access to values of datons by using hashing on the name of the daton. For each daton, it maintains its status ("being computed", "computed") and a list of demands of that value in the form of return addresses in the term store. The value store also determines when a computed daton value is no longer needed using a deterministic storage management scheme or a heuristic.

The interface is an input/output processor which allows user to communicate using demands for outputs and receive demanded outputs.

The processor pool consists of a set of processors which perform the Lucid arithmetic and logic operations efficiently.

Such a computer will not exploit all of the inherent parallelism in a program. However, some parallelism will be exploited because each of the components can be simultaneously active. That is, the term store could be creating new demands while the processor pool is
executing several operations. At the same time, the value store could be handling a demand for a daton value.

A Multi-board Parallel Computer

It is possible to use the single-board pipelined computer in designing a parallel implementation. This implementation consists of several boards each of which is connected using the interface (which needs to be modified) to an staged interconnection switch. The switch allows communication between any two boards via their interface. Figure 4.4 shows the architecture of the parallel computer.

Parallelism inherent in a program is exploited in the following manner. When a daton-value of a variable is demanded at a term store, it is not necessarily sent to the local term store. Instead, the daton name is used to uniquely select another board and the demand for daton value is sent to the value store on that board using the interconnection switch. Further processing of the demand is conducted at the destination board which may further result in demands for other datons of variables to be processed on other boards. Note that when a daton-value of a term is demanded at a term store, it is processed locally.

Therefore, distribution of computation is performed by allocating activities to compute daton values of variables onto different boards. The efficacy of this distribution depends on how uniformly daton names can be mapped to the different boards of the parallel computer.
5. Conclusion

We have demonstrated, using a Lucid program for the n-body problem, that it possible to write programs for parallel processing without having to say explicitly what is to be done in parallel and what is not. The declarative nature of an implicitly parallel language such as Lucid, resulting, as it does, in properties such as referential transparency, allows for rapid prototyping and modifiability. Moreover, if the language is intensional, like Lucid, programs are very concise, and can read much like the equations in physics they are intended to solve (if that is the sort of application being considered). This means that the programmer can concentrate on the application (the physics, say), rather than the details of managing multiple processors.

The parallelism inherent in programs in languages like Lucid can be extracted by using an appropriately designed architecture, or by implementing the language on one of the current crop of commercially available multiprocessor systems. The latter approach has limitations, because the implicit parallelism in programs that can best be discovered automatically is often of a fine-grain variety. Most of the commercially available multiprocessor architectures are really ensembles of von Neumann machines which makes this type of parallelism more difficult to exploit in the sense that the compiler technology to do so is not very mature. If we take a prescriptive view of the situation we can see that architectures can be designed to handle languages of the type we have been considering. In this paper we argue that there is an architecture, the Eduction Evaluator, that can exploit the parallelism implicit in programs such as the n-body program discussed earlier.

6. References

[Ashcroft 86]
E.A. Ashcroft,
_Dataflow and Eduction: Data-driven and Demand-driven Distributed Computation_,

[Faustini 87]
A.A. Faustini and W.W. Wadge
_Intensional Programming_
The Role of Languages in Problem Solving 2
J.C. Boudreaux, B.W. Hamill, and R. Jernigan (Editors)
Elsevier Science Publishers B.V. (North-Holland), 1987

[Fox 88]
G. Fox et al.
_Solving Problems on concurrent processors: Volume 1_
Prentice-Hall 1988

[Jagannathan 88]
R. Jagannathan,
_A Descriptive and Prescriptive Model for Dataflow Semantics_,
[Peyton Jones 87]
Simon L. Peyton Jones
The Implementation of Functional Programming Languages
Prentice-Hall Series in Computer Science, 1987

[Treleaven 82]
P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins,
Data-driven and demand-driven computer architecture,
ACM Computing Surveys,

[Wadge 85]
W.W. Wadge and E.A. Ashcroft,
Lucid, the Dataflow Programming Language,