An introduction to the data-parallel paradigm

Jonathan M. D. Hill
The London & South-East centre for High Performance Computing,
Queen Mary & Westfield College,
University of London.

Abstract

A data-parallel style of programming is one method of utilising the enormous computing power available on today's parallel machines. It is becoming clear that the data-parallel paradigm is not tied to a particular "flavour" of parallel architecture, be it Single Instruction Multiple Data-stream (SIMD) or Multiple Instruction Multiple Data-stream (MIMD). The purpose of this paper is to give an introduction to the basic techniques that form the building blocks of a data-parallel paradigm. The combinatorial search problem of finding the shortest path between two nodes in a graph is investigated, and two parallel solutions are presented. The first is based upon the scan algorithm that forms the heart of many data-parallel applications; the second uses a technique from the parallel solution of a computational fluid dynamics problem. The examples demonstrate that a parallel solution to a given problem may not necessarily bear any resemblance to a sequential solution—although the parallel techniques used here can be re-applied in a variety of problem areas.

1 Introduction

"One inconvenient thing about a purely imperative language is that you have to specify far too much sequencing. For example, if you wish to do a matrix multiplication, you have to do $n^3$ multiplications. If you write an ordinary program to do this, you have to specify the exact sequence which they are all to be done. Actually, it doesn't matter in which order you do the multiplications so long as you add them together in the right groups."—C. Strachey, 1966 [15].

All models of parallel computation arise from the observation that there is an inherent lack of sequencing in many problems. In a message-passing system this lack of sequencing is utilised by decomposing a problem into distinct tasks that are evaluated concurrently; communication between these tasks enables the cooperative solution of a problem. In contrast to a message-passing approach, a data-parallel paradigm is concerned with the parallelism that arises from the regular manipulation of the elements of large monolithic data-structures such as arrays. From an architectural standpoint, as opposed to manipulating single words of a machines memory in a sequential microprocessor, a data-parallel machine operates on a collection of

---


Email: J.M.D.Hill@lpc.ac.uk, WWW: http://www.lpc.ac.uk/SEL-HPC/People/Hilly/
words in parallel. This in-turn has an impact on a data-parallel language such that a collection of words constitute a data-parallel object in which each word can be interpreted as the contents of one of the processors of a data-parallel machine. When programming in such a language, two techniques are often used to utilise the lack of sequencing in problems:

- If the order of evaluation of a sequence of similar operations is irrelevant, then they can be applied synchronously and in parallel all at the same time. The idea behind a data-parallel language is to replace collections of similar operations by a monolithic operation that encapsulates the collection.

- General patterns of sequencing that do not fall into the “all at once” category above can be parallelised by using techniques that exploit the sequencing that arises from algebraic considerations such as associativity\(^1\) and commutativity\(^2\).

The rest of the paper investigates these two techniques. Initially, §2 builds upon the first technique of “all-at-once” computations by considering a correspondence between simple for-loops and data-parallel equivalents in Fortran 90. The second technique of using algebraic properties is investigated in the context of parallelising the summation of a array of numbers in §3. The specific technique used in such a summation is to apply a parallel scan to calculate all the partial sums of an array. The scan algorithm is considered abstractly in §3.3, and is used to redefine some of Fortran 90’s intrinsic functions, as well as being used in the parallel solution of finding the shortest path between two nodes within a graph in §4. The paper is concluded in §5 by considering an alternative approach to a similar searching problem using a technique from computational fluid dynamics. One of the benefits of the latter parallel solution is that it allows searching for the shortest path through a dynamically changing graph.

2 All at once computations

Figure 1 shows a Fortran 90 for-loop that ‘safely’ divides each element of the one-dimensional array arrA by corresponding elements from arrB (the arrays are assumed to be of the same size and bounds). The semantics (i.e., meaning) of a loop in an imperative language such as Fortran 90 imposes far more sequencing on the problem than is necessary. Therefore the implied semantics are that the loop starts at an index that represents the lower-bound of the two arrays (calculated by the lbound function) and during successive iterations pairs of elements from both arrays with the same index-value are divided one by the other. In this example the semantics of the loop would be unaltered if the loop started iterating from the opposite end of the two arrays and worked downwards. More importantly, an alternative parallel strategy could be used where each of the divisions in the body of the loop could occur synchronously and in parallel all at the same time.

Such an “element-wise”, “map”, or “all-at-once” operation on arrays is the essence of a data-parallel paradigm. The benefits of parallelising loops in such a manner is that if the loop has a range over \(N\) values, then instead of executing order \(N\) sequential iterations which gives the loop a \(O(N)\) complexity, a data-parallel implementation only requires a single parallel implementation.

\(^1\)A binary operator \(\oplus\) is associative if \(x \oplus (y \oplus z) = (x \oplus y) \oplus z\).

\(^2\)A binary operator \(\oplus\) is commutative if \(x \oplus y = y \oplus x\).
\begin{lstlisting}
DO i = lbound(arrA), ubound(arrB)
   IF (arrB(i) /= 0) THEN
      result(i) = arrA(i) / arrB(i)
   END IF
END DO

FORALL (i = lbound(arrA):
   ubound(arrB),
   arrB(i) /= 0)
   result(i) = arrA(i) / arrB(i)
END FORALL
\end{lstlisting}

Figure 1: Fortran 90 element-wise division

WHERE (arrB(:) /= 0)
   result(:) = arrA(:) / arrB(:)
END WHERE

Figure 2: HPF element-wise division

Figure 3: Fortran 90 monolithic divide

step in which all the iterations of the loop occur concurrently. Such a parallel loop has a $O(1)$ complexity.

Instead of writing for-loops and allowing a compiler to infer whether they exhibit the “all-at-once” property as vectorising compilers do, the approach adopted by Fortran 90 and its data-parallel successor High Performance Fortran\(^3\) is to provide a syntactic framework within which the programmer can express all-at-once loops. We consider two different ways of achieving this in HPF:

1. A parallel for-loop is used in preference to a sequential for-loop. The parallel loop has a semantics that relies upon there being no implied sequencing between iterations of the loop. This enables the iterations to be evaluated in any order, including the evaluation of all the iterations at the same time. The trade-off of such a loop is that it shifts onto the programmer the burden of proving that there is no implied sequencing.

2. Large monolithic operations that encapsulate for-loops are used instead. For example, a data-parallel analogue to the addition operation is a data-parallel addition that adds element-wise positions of two arrays together in parallel.

Figure 2 shows an example of the first approach where the loop of figure 1 is translated into a parallel FORALL loop. The header of the FORALL contains a triplet of the form:

\begin{verbatim}
FORALL (index-name = subscript : subscript, [: stride] [, mask-expr])
\end{verbatim}

that specifies a mask as well as the range of values over which the loop iterates. The \textit{index-name} specifies a variable that is instantiated to the values in the range \textit{subscript} to \textit{subscript} in steps of \textit{stride} during the different iterations of the loop. Unlike an ordinary loop, the FORALL contains an optional guard \textit{mask-expr} such that specific iterations of the loop only occur when the guard is true. Notice how the transformation from the loop in figure 1 into the one in figure 2 is syntactically superficial. However, the user has to be aware of various restrictions on the type of statements that are allowed in the body of the loop\(^4\):

\(^3\)HPF is an extension to Fortran 90 that facilitates data-parallel programming. The differences between Fortran 90 and HPF are that the latter has a parallel FORALL loop; directives that give hints to the compiler to facilitate the mapping of arrays onto the processors of a multi-processor machine; an extensive data-parallel library of functions.

\(^4\)More details on the restrictions on FORALL statements can be found in [13, page 171]

3
• Conditional statements are not allowed within the body of a parallel FORALL. This is
one of the reasons for the mask expression in the header of the FORALL which enables a
conditional such as the one shown in figure 1 to be "moved outside" the loop body.

• Any variables used on the left-hand side of an assignment in the body of the loop
cannot be used on the right-hand side. This requirement ensures that the loop has the
all-at-once property.

• Any function used as either the mask to a FORALL, or within the body of a FORALL
must have the PURE annotation to ensure the function doesn’t side-effect any global
variables making the all-at-once computation of the for-loop meaningless. For example,
if a function is evaluated in parallel at each iteration of a FORALL, and that function
modifies a global variable, then the contents of the variable on completion of the loop
will be undefined\(^5\).

An alternative approach to the parallelisation of the loop of figure 1 is shown in figure 3.
The idea is to replace for-loops by monolithic operations that encapsulate the loop. The
transformation is rather mechanical, and is guided by the goal of transforming complex loops
into collections of simpler loops that contain primitive operations\(^6\). For-loops containing
primitive operations are then replaced by monolithic parallel operations (a formal account of
a similar transformation can be found in [9]). For example, a simple loop iterating over two
arrays adding element-wise index-values together would be replaced by a monolithic array-
addition operation. The key to parallelising for-loops by translating to monolithic operations
is to be aware of the parallel analogues to the sequential operations and control structures of
Fortran 90. The parallel analogue to integer division written as \(/\), is a parallel division also
written as \(/\) (similar analogues exits for the other mathematical operations). Although the
overloading of arithmetic operations simplifies the presentation of algorithms, it requires the
user to be aware of the context of the operation to make the distinction between an array or
scalar operation—this is similar to the problem of distinguishing between an integer or real
arithmetic operation. To make the presentation of figure 3 clearer, the redundant annotation
(\(\cdot\)) on variables is used as a reminder that an array contains many values.

As well as monolithic analogues to the arithmetic operations, the analogue to the
conditional "if-then-else" statement is a parallel WHERE. The semantics of a WHERE is that the
conditional represents an array of logicals such that wherever there is a true, then any array
computations in the body of the WHERE only occur at similar array index-values. Therefore
the program in figure 3 is interpreted as:

1. The conditional of the WHERE is evaluated. As ARR B represents an array then \(=/\) is the
monolithic array not-equals comparison and zero represents an array the same size and
rank as ARR B which contains zero at every index.

2. Evaluation of the conditional creates a logical mask such that wherever there is a true
(i.e. where ARR B is not zero) then similar index values of any arrays in the body of the
WHERE are evaluated.

3. The divisions of ARR A by ARR B occur in parallel at index-values defined by the mask of
the WHERE; RESULT is only defined at similar locations.

\(^{5}\)This is the same problem as multiple writers changing a shared variable in a concurrent system.

\(^{6}\)The inverse of the loop fusion [1] compiler optimisation is performed.
FUNCTION PartialSums(array)
    INTEGER, intent(IN) :: array(:)
    INTEGER, dimension(size(array)) :: PartialSums
    INTEGER :: i

    PartialSums(1) = array(1)
    DO i = 2, size(array)
        PartialSums(i) = PartialSums(i-1) + array(i)
    END DO
END FUNCTION PartialSums

Figure 4: Sequential solution to the partial-sums problem

In the rest of the paper, we show how all-at-once computations form the heart of more advanced data-parallel algorithms. In particular, either way of expressing the all-at-once computations outlined above can be used as part of the parallel summation problem in the next section.

3 Utilising associativity: deriving a parallel scan function

Scan is widely accepted as a fundamental technique of a data-parallel paradigm. Its uses have been as diverse as sorting [2, 10], line drawing [2], normal order reduction of the λ-calculus [17], lexical analysis [19, 7], word searching [6, 9], implementing iterative constructs such as for-loops [11, 12], and LL(1) parsing [6, 7] to name but a few. A step-by-step guide to the parallel implementation of scan is presented. First the associated problem of evaluating the partial sums of the series \( x_0, \ldots, x_n \) defined by equation (1) is explored. The technique is then generalised to form the definition of scan.

\[
\text{partialSums}([x_0, x_1, \ldots, x_n]) = [x_0, \sum_{i=0}^{1} x_i, \sum_{i=0}^{2} x_i, \ldots, \sum_{i=0}^{n} x_i]
\]  

(1)

3.1 Sequential: the partial sums of an array

Since the partial sums function computes a series of numbers, it is natural to represent this series as an array. By reading equation (1) denotationally we can inductively define the partial sums function in terms of a for-loop which writes its results into an array (see figure 4). For a given iteration \( k \) of the loop, the partial sums of the preceding \( k - 1 \) elements will be stored within the \( k - 1 \)th element of the array PartialSums. The \( k \)th partial sum can therefore be calculated by adding the sum of the \( k - 1 \) elements to the \( k \)th element from array. As the loop performs order \( N \) iterations (where \( N \) is the size of array) the algorithm has a \( \mathcal{O}(N) \) complexity.

3.2 Parallel: the partials sums of an array

As the PartialSums function only contains a simple for-loop, it would be expected that the function could be parallelised by replacing the for-loop by a parallel FORALL such as the one
used in §2. Unfortunately such a translation is semantically incorrect as one of the side-conditions on the use of a parallel FORALL will be violated as the same variable is used on both the left and right-hand sides of an assignment in the body of the loop.

The example demonstrates the need for such side-conditions since the algorithm is inherently sequential as the solution of the $k^{th}$ partial sum directly depends upon the $k - 1^{st}$ partial sum, which in-turn depends upon the $k - 2^{th}$, etc. The first step towards a truly parallel implementation is attained by breaking this dependence.

A similar problem to that of generating the partial sums is shown in figure 5(a), where the sum of a one-dimensional array of numbers (the circles represent processor/array elements) is achieved by a divide and conquer technique. The dependence between consecutive elements of the array has been broken by utilising the associativity of addition. An informal justification for the technique is that given an array containing four values [$x_1$, $x_2$, $x_3$, $x_4$], then the contents of the last element after applying the partial-sums function will be:

$$((x_1 + x_2) + x_3) + x_4$$

As addition is associative, the bracketing of the expression can be re-ordered so that instead of bracketing each of the expressions above in a leftwards manner, a rightwards bracketing can be used:

$$x_1 + (x_2 + (x_3 + x_4))$$

More importantly in a parallel setting, a tree-like bracketing structure can be adopted in which the implied sequencing of the addition has been broken, and the different groups of expressions can be evaluated concurrently, i.e.,

$$(x_1 + x_2) + (x_3 + x_4)$$

If we were implementing this divide and conquer algorithm on a data-parallel machine, we would impose a structure on the combination of processors such that all the combinations at each level of the evaluation tree shown in figure 5(a) occur synchronously, and in parallel. Therefore the addition of the group containing one and two can occur concurrently with the group containing three and four. Another problem with the tree shown in figure 5(a) is that each of the circles represent array elements which are an abstraction of the processing elements of a data-parallel machine. The way in which communication is typically utilised in a data-parallel language is that the data stored in the processors which wish to communicate with each other are arranged into a common array index-value. This style of communication is applied in the summation problem by skewing the tree as shown in figure 5(b), and the data-rearrangement is performed by a shift. As can be seen from the figure, the last element...
FUNCTION PartialSums(array)
    INTEGER, intent(IN) :: array(:)
    INTEGER, dimension(size(array)) :: PartialSums
    INTEGER, dimension(size(array)) :: arr
    INTEGER :: k=1, x

    arr(:) = array(:)
    DO WHILE (2***(k-1) <= size(array))
        WHERE ((x >= 1 + 2***(k-1), x=1, size(array))/)
            arr(:) = arr(:) + eoshift(arr(:), SHIFT=-2***(k-1))
        END WHERE
        k = k + 1
    END DO
    PartialSums(:) = arr(:)
END FUNCTION PartialSums

Figure 6: Parallel solution to the partial-sums problem

of the array contains the sum of the array, yet the first two array elements also contain their partials sums. The key to producing a parallel partial sums function is to utilise all of the processors during each iteration of the divide and conquer algorithm.

Figure 5(c) shows such an algorithm which uses the parallel prefix technique [14, 2, 19, 8]. We can generalise from this figure that during the \( k \)th iteration of the algorithm (where \( k \geq 1 \)) a subset of the processing elements combine with a processor \( 2^{k-1} \) places to their left. If \( \text{arr} \) is the state of the array during the start of each iteration, then we can express such a combination using the Fortran 90 intrinsic \( \text{eoshift} \):

\[
\text{arr}(:) + \text{eoshift(} \text{arr}(:), \text{SHIFT}=-2**\text{(k-1)})
\]

Next, if arrays are assumed to have a lower bound of one then the array elements in the range 1 to \( 2^{k-1} \) do not combine with any processors and have there partials sums defined. Similarly the range of processors that combine with processors during the algorithm can be expressed by the Fortran 90 array comprehension:

\[
((x \geq 1 + 2**\text{(k-1)}, x=1, \text{size(array)})/)
\]

Finally, we define the parallel partial sums function in figure 6 by iterating the combining process a logarithmic number of times, where the mask is used to select those processors which are involved in the combination using a Fortran 90 \text{where} statement. As there are a logarithmic number of iterations each performing a \( \mathcal{O}(1) \) complexity element-wise addition, then the entire algorithm's complexity is \( \mathcal{O} (\log N) \) for an input of size \( N \).

3.3 Higher order glue: \textit{scan} and \textit{fold}

Examining the definition of the sequential and parallel partial sums function, we see that only + is specific to the computation—i.e., if we wanted to calculate the partial-products then we would replace + by \#. One way of performing such a substitution is to parameterise the partials sums function on the function applied at each iteration of the partials-sums function, defining \textit{Scan} as shown in figure 7. It should be noted however that the + used in the definition is the all-at-once array version of the addition operator. Therefore when the programmer uses \textit{scan},
FUNCTION Scan(fun, array)
    INTERFACE
    PURE FUNCTION fun(x, y)
    INTEGER, intent(IN) :: x(:), y(:)
    INTEGER, dimension(size(x)) :: fun
    END FUNCTION fun
    END INTERFACE
    INTEGER, intent(IN) :: array(:)
    INTEGER, dimension(size(array)) :: arr
    INTEGER :: k = 1, x
    arr(:) = array(:)
    DO WHILE (2**k-1 <= size(array))
        WHERE (((x >= 2**(k-1), x = 1, size(array))/))
            arr(:) = fun(arr(:), &
                eoshift(arr(:),SHIFT=-2**(k-1)))
    END WHERE
    k = k + 1
    END DO
    Scan(:) = arr(:)
END FUNCTION Scan

Figure 7: Parallel definition of scan

FUNCTION Fold(fun, array)
    INTERFACE
    PURE FUNCTION fun(x, y)
    INTEGER, intent(IN) :: x(:), y(:)
    INTEGER, dimension(size(x)) :: fun
    END FUNCTION fun
    END INTERFACE
    INTEGER, intent(IN) :: array(:)
    INTEGER :: Fold
    INTEGER, dimension(size(array)) :: temp
    temp(:) = Scan(fun, array)
    Fold = temp(size(array))
END FUNCTION Fold

Figure 8: Parallel definition of Fold

they will have to guarantee that the function passed as an argument to scan is the all-at-once version of a binary associative operator to ensure that the algorithm is well defined.

Having produced a parameterised version of scan, its definition can be re-used in an implementation of fold\(^7\) that calculates the value produced in the last array element of the scan. Given the definitions of fold and scan, a variety of Fortran 90’s intrinsic functions can be redefined in terms of the following equational relationships\(^8\):

\[
\begin{align*}
\text{SUM}(\text{array}) &= \text{Fold}(+, \text{array}) \\
\text{PRODUCT}(\text{array}) &= \text{Fold}(*, \text{array}) \\
\text{MINVAL}(\text{array}) &= \text{Fold}(\text{elementwiseMin}, \text{array}) \\
\text{MAXVAL}(\text{array}) &= \text{Fold}(\text{elementwiseMax}, \text{array})
\end{align*}
\]

where \text{elementwiseMin} (see figure 10) is the all-at-once version of the \text{min} function (see figure 9) on integers.

4 Calculating the shortest path with a scan

The purpose of this section is to define a parallel graph searching algorithm that finds the shortest path between any two nodes within a directed graph. Figure 11 shows an example directed graph in which the capital letters represent nodes, and the labelled arrows represent a transition between two nodes. Before introducing the parallel graph searching algorithm,

\(^7\)The term fold can be understood in terms of folding an operator between each element in a sequence of values to create an expression that when evaluated reduces the sequence to a single value.

\(^8\)Fortran 90 definitions are not given to save space.
some basic set-theoretic definitions are given, which will be used throughout the following discussion.

4.1 Set theory definitions

A set is an un-ordered collection of values that doesn’t contain any duplicates. The set $\mathbb{Z}$ is the set of positive and negative integers written as $\{0,1,-1,2,-2,\ldots\}$. A Zermelo-Fraenkel set expression [20] that denotes the intersection of two sets $A$ and $B$, written as $A \cap B$ can be expressed as $\{x \mid x \in A \land x \in B \}$. The repeated variables within the expression are used to impose a constraint on the two sets $A$ and $B$, such that the $x$ in both the $\in$-expressions denote the same value. For two sets $A$ and $B$, the Cartesian or cross product of $A$ and $B$, denoted by $A \times B$ is the set of pairs $\{(a,b) \mid a \in A \land b \in B\}$.

Definition 1 (relation) For any sets $A$ and $B$, any subset of $A \times B$ is called a relation from $A$ to $B$.

Definition 2 (relational composition) Given two relations, $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, the composition of the two written as $(R_2 \circ R_1)$ is a relation from $A$ to $C$ defined by:

$$R_2 \circ R_1 = \{(x,z) \mid (x,y) \in f \land (y,z) \in g \}$$

4.2 Representing graphs as relations

The graph of figure 11 can be represented by the relation $G = \{(A,B),(A,C),(B,C),(C,D)\}$, where the source of the relation represents all the nodes where transitions start from, and
the range represents the destination nodes of the transitions. Another way of interpreting such a relation is that it represents all the paths in the graph of length one. Therefore if $G$ is composed with itself, then the relation $G \circ G$ represents all the paths of length two. A parallel algorithm for finding the shortest path between nodes $x$ and $y$ in the graph $G$ can be calculated by combining $G$ with itself until the relation

$$G^k = G \circ \cdots \circ G$$

contains a transition between $x$ and $y$; the shortest path will be of length $k$.

4.3 The details

A parallel implementation of the graph searching problem starts by taking a guess at a path length that is definitely longer than the shortest path between the nodes that are to be investigated. If the guess is $N$, then the relational representation of the graph is initialised within $N$ elements of an array. For the relation of figure 11, and a value of $N = 4$, then the first box in figure 12 shows the initialised state.

As relational composition is associative\(^9\) we define a function $\text{elementwiseCompose}$ to be the all-at-once version of $\circ$, and scan the function across the initialised sequence of relations. Figure 12 shows each of the logarithmic number of steps of the parallel prefix technique used in the definition of scan. The shaded transitions shown in figure 12 represent the path identified by the bold arrows in the directed graph also shown in the figure. The problem of finding the shortest path between nodes $x$ and $y$ is solved by finding the first processor with the transition $x \rightarrow y$ within it. The algorithm shown diagrammatically in figure 12 assumes that the pairs of nodes in the relation are also annotated with an identifier that represents the transition between the two nodes. If these identifiers are concatenated during relational composition, then on termination of the algorithm the annotations will define the shortest path.

\(^9\)i.e., $(G \circ G) \circ G = G \circ (G \circ G)$
FUNCTION FirstTrue(array)
    LOGICAL, intent(in) :: array(:)
    INTEGER :: FirstTrue
    INTEGER, dimension(size(array)) :: index
    INTEGER :: i

    WHERE (array)
        index(:) = ((i,i=1,size(array))/)
    ELSEWHERE
        index(:) = size(array)
    END WHERE
    FirstTrue = Fold(elementwiseMin,index)
END FUNCTION FirstTrue

Figure 13: Finding the index of the first true in an array of logicals

All that remains is to define a parallel algorithm for finding the first occurrence of a true within an array of logicals which can be used to identify the transition from the source to destination state that represents the shortest path within a graph. Figure 13 shows a $O(\log N)$ complexity algorithm that solves the problem, where the parallelism can be attributed to parallel fold.

4.4 Is the technique appropriate?

The key to successful parallel programming is to be aware of the point within an algorithm when parallelism becomes worthwhile. The parallelism within the graph algorithm can be attributed to parallel scan. For scan to be effective, the size of the array being scanned must be large (i.e., thousands of elements), and as a consequence searching only becomes effective if the shortest path is relatively long, which in-turn implies that the graph being traversed is large. Scan has a logarithmic complexity if the logarithmic number of element-wise combinations of the associative operator being scanned has a $O(1)$ complexity. In the graph algorithm, as relational composition works on a dynamically changing data-structure, which potentially squares in size after each composition step, the worst case complexity of the entire algorithm is $O(2^{\log^N})$. The reason for this is that the algorithm is in effect performing a breadth first search, which is susceptible to the intermediary relations becoming very large within a few composition steps. This contention between length of the shortest path and the relation size is highlighted when the graph searching is used to solve the knights tour problem [3].

A knights tour of a chess board is the trail of successive knights moves, such that starting at any square on the board, the knight visits every square on the board once, finishing back at the square the tour started. Figure 14 shows a tour for a 8 x 8 chess board. Note that the tour starts in the top left hand corner of the board and finishes at a position where the first square is reachable.

The ways in which a knight can possibly move around a chess board can be considered as a graph which can be more easily understood in terms of a relation containing all knight-like moves of length one. For example, the relation for the transitions of the top-left most square on the board (where numbering starts from the top left-hand corner working along the rows) is $K_1 = \{(1,11),(1,18)\}$. As there are only 64 possible moves in a knights tour, parallelising
the search with a scan probably isn’t worthwhile. In the context of the knights tour one fruitful way of parallelising the problem would be to parallelise the Cartesian product used in the functional composition as the intermediary relation is potentially enormous. In general, to make the searching algorithm more effective, heuristics are required to minimise the size of the intermediary relations. These heuristics will be problem dependent, and are outside the scope of this paper.

5 Searching using computational fluid dynamics

A similar problem to the graph searching described in §4 is the navigation of a mobile robot through a two-dimensional space. The robot is assumed to possess a detailed plan of its environment, and the problem is to find the shortest path to any point within the environment.

Classically [16] a robot path planning algorithm constructs a graph where the nodes represent the obstacles in the environment, and the edges connect nodes which correspond to obstacles which can “see” each other. When the navigation problem has been converted into such a form, a parallel searching algorithm such as the one described in §4 can be used. The major drawback of this approach is if the environment is dynamic, then the graph will need to be recomputed, which may be time consuming for realistic environments.

Decuyper [4] proposes a navigation algorithm based upon the physical conditions of fluid dynamics where the movement of a robot is modelled by a particle flowing through a system. The fluid dynamics metaphor is utilised by creating a flow from the robot to its destination which the robot consequently follows.

Figure 15 shows the robots environment modelled in terms of fluid flow. A “water tap” is used to represent the robots position, and a “plug-hole” its final destination. If the environment is initially filled with water, a flow is created between the tap and plug-hole, and the pressure gradient produced as a result of the flow will be the minimal path between the robot and its destination.

The elegance of this technique is that it can easily cope with dynamic environment changes (i.e., a wall collapsing), because the fluid flow continually changes to take into consideration the new environment.

The relationship between this analogue technique and parallel processing is that the physical problem of fluid flow has undergone enormous research efforts over the last decade, with many parallel implementations to the problem on a variety of parallel architectures (see [18, 5]
for more details). The important feature of this parallel technique is that the searching problem can be made architecture independent by building it on-top of a CFD implementation that can be tailored to a particular parallel architecture. In contrast to this rather limited form of architecture independence, the all-at-once and scan type computations are truly architecture independent as has been identified in the High Performance Fortran language definition.

6 Conclusions

The purpose of this paper has been to show that a parallel solution to a problem may be radically different than a classical solution for a sequential microprocessor. By presenting general data-parallel techniques that can be reused in a variety of problem areas, the aim has been to introduce a data-parallel way of approaching problems so that algorithms such as the ones presented here no longer look strange.

7 Acknowledgements

I would like to thank my colleagues at the London Parallel Applications Centre (LPAC) for reading earlier drafts of this paper. In particular John Steele, Heather Liddell, Dimitris Tsaptsinos, Carlos Palko Korn, Alistair McEwan, and Kevin Walters.
References


Hello Barry,

Could you give me a tech report number for the following paper?

Cheers....

jon

An introduction to the data-parallel paradigm

Jonathan M. D. Hill

Abstract:

A data-parallel style of programming is one method of utilising the enormous computing power available on today's parallel machines. It is becoming clear that the data-parallel paradigm is not tied to a particular 'flavour' of parallel architecture, be it Single Instruction Multiple Data-stream (SIMD) or Multiple Instruction Multiple Data-stream (MIMD). The purpose of this paper is to give an introduction to the basic techniques that form the building blocks of a data-parallel paradigm. The combinatorial search problem of finding the shortest path between two nodes in a graph is investigated, and two parallel solutions are presented. The first is based upon the scan algorithm that forms the heart of many data-parallel applications; the second uses a technique from the parallel solution of a computational fluid dynamics problem. The examples demonstrate that a parallel solution to a given problem may not necessarily bear any resemblance to a sequential solution---although the parallel techniques used here can be re-applied in a variety of problem areas.

Keywords:

Data-parallelism; all-at-once computations; map; fold; scan; graph searching; computational fluid dynamics; Fortran 90; High Performance Fortran.