Speed up and Efficiency of Parallel Algorithms

Arithmetic Expression Evaluations

Vector and Matrix Algorithms

Speedup and Efficiency of Algorithms

For any given computation (algorithm):

Let T_P be the time to perform a computation with P processors.(arithmetic units, or PEs)

We assume that any P independent operations can be done simultaneously.

Note: The depth of an algorithm T_∞ , the minimum execution time.

The speedup with P processors is $S_P = \frac{T_1}{T_P}$, and efficiency is

$$E_P = \frac{S_P}{P}$$

These numbers refer to an algorithm and not to a machine. Similar numbers can be defined for specific hardware.

The time T_1 can be chosen in different ways:

To evaluate how good an algorithm is, it should be the time for the "BEST" sequential algorithm.

<u>The Minimum Number of Processors Giving the Maximum Speedup:</u>

Let Π be the min. number of P of processors such that $T_P = T_\infty$ i.e.

$$\Pi = \min \{ P \mid T_P = T_\infty \}$$

Then T_Π, S_Π, E_Π are the best known time, speedup, and efficiency respectively.

EXAMPLES:

For the Upper/Lower parallel prefix algorithm:

Π **($P^{ul}(N)$) = $\lfloor N/2 \rfloor$, (number of parallel operations in each level)**

T_{Π} ($P^{ul}(N)$) = $\lceil \log_2 N \rceil$, Depth

For the Odd/Even parallel prefix algorithm:

Π **($P^{oe}(N)$) = $\lfloor N/2 \rfloor$, (number of parallel operations in first level)**

T_{Π} ($P^{oe}(N)$) = $2\lceil \log_2 N \rceil - 2$, Depth

Speedup and Efficiency of Upper/Lower

Since $P^{ul}(N)$ has $\lfloor N/2 \rfloor \lceil \log_2 N \rceil$ operations, we might write speedup as:

$$S_{\Pi}\left(P^{ul}(N)\right) = \frac{\left\lfloor \frac{N}{2} \right\rfloor \lceil log_2 N \rceil}{\lceil log_2 N \rceil} = \frac{T_1}{T_P} = \left\lfloor \frac{N}{2} \right\rfloor$$

This would make efficiency :

$$E_{\Pi} = \frac{S_{\Pi}}{P} = \frac{\left\lfloor \frac{N}{2} \right\rfloor}{\left\lfloor \frac{N}{2} \right\rfloor} = 100\%$$

BUT, this means executing a parallel algorithm of large size with one processor.

If we have one processor, we should use the smaller sequential algorithm

$$S_{\Pi}\left(P^{ul}(N)\right) = \frac{T_1(sequentialP(N))}{T_{\Pi}\left(P^{ul}(N)\right)} = \frac{N-1}{\lceil log_2 N \rceil}$$

This makes the efficiency,

$$E_{\Pi} = \frac{N-1}{\lceil log_2 N \rceil} \times \frac{1}{\left\lfloor \frac{N}{2} \right\rfloor} \cong \frac{2}{log_2 N} \quad , \qquad for N \geq 4$$

Thus the efficiency of Upper/Lower really decreases as the problem size grows.

The Evaluation of Arithmetic Expressions

Most problems are not so simple that best sequential algorithm is known, to say nothing of the best parallel algorithm.

Arithmetic expression evaluation is a case in which general results are known:

Definition:

An atom is a constant or variable appearing in an expression.
Let E<N> be an expression in +, -, ×, /, (,) having N atoms.
The minimum time for sequential evaluation of any such algorithm is N-1 steps.

Evaluation of E<N> must take a minimum amount of time as shown by the lemma.

Lemma 1:

For any number P of processors, the time to evaluate an expression in N atoms satisfies:

$$T_P(E\text{<N>}) \geq \lceil log_2 N \rceil$$

The proof is based on the fact that all atoms are combined into one result.

Proof:

Since there is a single result and all operations are dyadic (they take 2 operands), then there is only one result at the last step and no more than 2 intermediate results at the nest to the last step.
In general, there are at most 2^l intermediate values at the last-l step.

Since there must be N atoms at stage l, then $N \leq 2^{\text{no. of stages}}$ and

$$(T_P(E\text{<E>}) = K = \text{number of stages}) \geq \lceil log_2 N \rceil$$

QED.

Expressions can be transferred by mathematical operations into more parallel forms.

Using associative, commutative, and distributive laws we can reduce the height of an expression tree.

Recall:

Commutativity: $X+Y = Y+X$ and $X*Y = Y*X$

Associativity: $(X+Y)+Z = X+(Y+Z)$ and $(X*Y)*Z = X(Y*Z)$

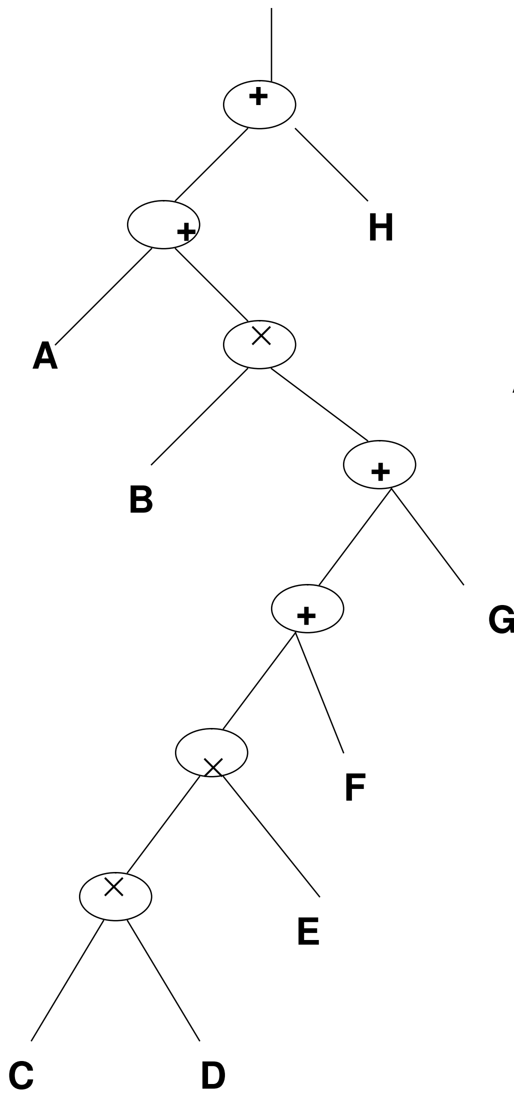
Distributivity: $X(Y+Z) = XY + XZ$ and $(X+Y)Z = XZ + YZ$

Consider the expression:

$$E\langle 8\rangle = A + B(CDE + F + G) + H$$

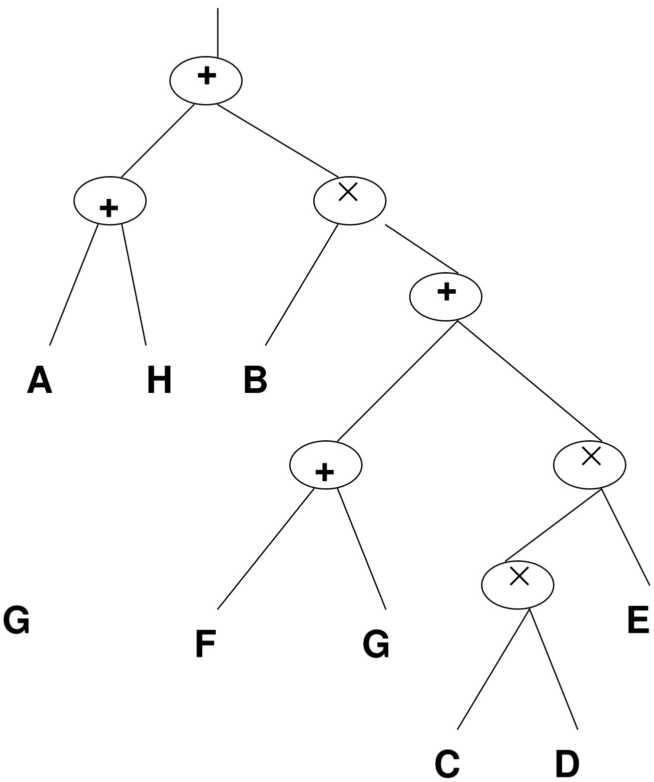
E<8> = A + B(CDE + F + G) + H

The evaluation order as written is given by the dependence graph



Depth = 7
Size = 7

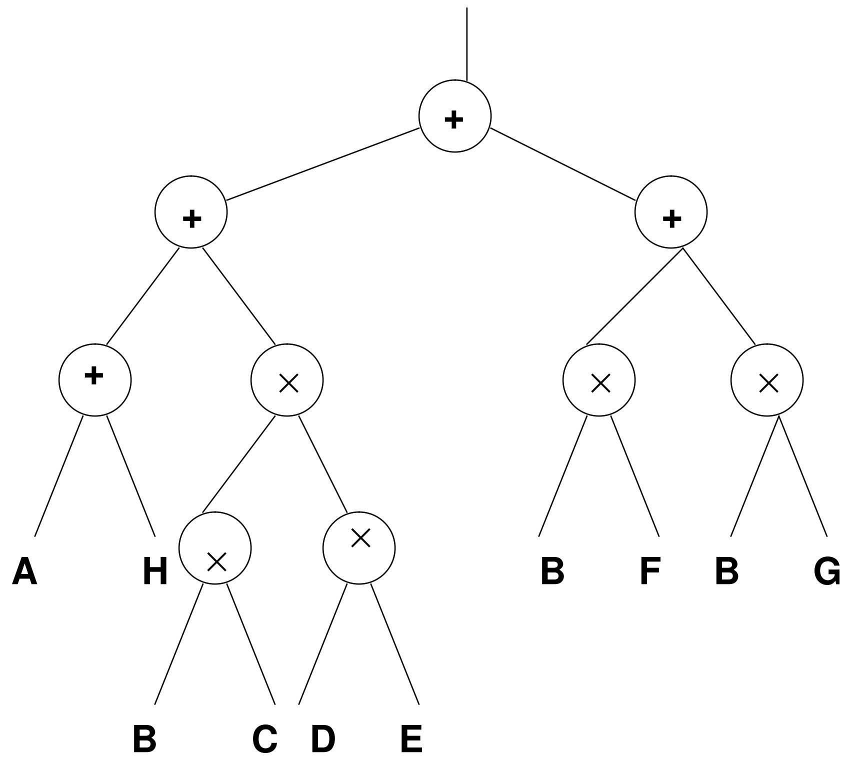
The expression can be transformed by associativity and commutativity into:



A+H+B(F+G+CDE)

Depth = 5
Size = 7

Including the distributive law, we can get an even smaller depth. But the number of operations will increase.



A+H+(BC)(DE)+BF+BG

Depth = 4
Size = 9

Using associativity and commutativity, evaluation of an expression is bounded.

In addition to lower bound on the evaluation time for arithmetic expression, we can also get an upper bound when associativity and commutativity are used to put the expression into the most parallel form possible:

<u>Theorem1</u>:

If E<N,d> is an arithmetic expression in N atoms with depth d of parenthesis nesting, then using commutativity and associativity only, E<N,d> can be transformed so that

$$T_{\Pi}(E<N, d>) \leq \lceil log_2 N \rceil + 2d + 1$$

with

$$\Pi \leq \left\lceil \frac{N}{2} \cdot d \right\rceil \qquad \text{processors.}$$

Reference:

J.L. Baer and D.P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computation," ***Proc. IFIP Congress 1968*****, North Holland, Amsterdam, pp. 340-346.**

If distributivity is also used, the upper bound is independent of parenthesis nesting.

But the size of the computation may increase, so the bound on Π also will increase.

<u>Theorem</u>:

An expression E<N> in N atoms can be transformed by associativity, commutativity, and distributivity so that

$$T_{\Pi}(E<N>) \leq \lceil 4 log_2 N \rceil \text{ , with } \quad \Pi \leq 3N \quad \text{processors.}$$

Note: The time bound only applies to transformed expressions. The transformation itself takes an order of $Nlog_2N$ steps.

For computations larger than single expressions, we must look at specific cases.

Vector and Matrix Algorithms

These algorithms have a specific property for parallel evaluation.

The same operation is applied to all components of vectors simultaneously.

Componentwise addition:

$$\vec{a} + \vec{b}: \qquad \mathrm{a}_1 + b_1, a_2 + b_2, \ldots, a_N + b_N$$

Another important property is the dot product. It is the componentwise multiplication followed by sum reduction over the products.

Reduction is a concept of more general use in vector computing. It can be done with any dyadic operator +, -, $\times$, •, max, min .

Reduction means writing the operator between all adjacent components and evaluating the arithmetic expression which results.

The APL language has a reduction operation:

$$\text{OP/V} = v_1 \text{ OP } v_2 \text{ OP } v_3 \text{ OP } \ldots\text{OP } v_N$$

If the operator is associative, then left to right, right to left, and tree evaluation orders all give the same mathematical results.

Matrix Vector multiplication

To illustrate some vector algorithm features:

$$\vec{Y} = A \cdot \vec{X}$$

Pseudo code for sequential algorithm :

```
For i:= 1 step 1 until N
        begin
        Y[0] := 0;
        For j:= 1 step 1 until N
            Y[i] := Y[i] +A[i,j]*X[j];
        end;
```

In this form the outer loop is over the rows of the matrix A, and the inner loop represents a dot product.

Another form of matrix-vector multiply arises from exchanging inner and outer loops:

```
For i:= 1 step 1 until N
        Y[i] := 0;
For j:= 1 step 1 until N
        For i := 1 step 1 until N
            Y[i] := Y[i] + A[i,j] * X[j];
```

In this form the outer loop is over columns of A. The inner loop multiplies all components of a column of A by one element of X and adds this vector of products to the partial result in Y.

The initialization of the dot products to zero (Y) before summing now is a separate loop at the beginning.

Matrix-Matrix Multiplication

Restructuring can also be done to the three nested loops of matrix-matrix multiply.

```
For i:= 1 step 1 until N
        For j:= 1 step 1 until N
            C[i,j] := 0;

For K:= 1 step 1 until N
        For i:= 1 step 1 until N
            For j:=1 step 1 until N
                C[i,j] := C[i,j] + A[i,k]*B[k,j];
```

This algorithm is called kji form, can be described as :
Initialize the result matrix elements to zero, and then add N by N matrices of product terms to form the result.

It is an alternative to the so called ijk form, in which the inner loop is over k and expresses the dot product of a row of A with a column of B:

The forme used as an example in SIMD pseudo code was the ikj form.