Parallel Vector Algorithms

1 Introduction

We will now study several algorithms where the parallelism can be easily expressed in terms of vector operations.

Simplistic timing figures will be given in some cases for pipelined machines, array machines, and multiprocessors.

In these timings, parallelism overhead, subscript computations, and memory access/communications costs will be ignored.

2 Time to execute a vector operation

Let us start with the simplest possible situation. Consider the following generic vector operation:

```
a(1:n) # b(1:n)
```

First, let us assume a pipelined arithmetic unit with s stages for operation #. Each stage takes τ units of time.

The time to execute the vector operation under these assumptions is :

$$t_{pipeline} = (s + (n-1))\tau$$

Compare this with the serial time when no pipelining takes place:

$$t_{serial} = s\tau n$$

Consider now an array machine with *P* arithmetic units, or a multiprocessor with *P* processors.

The execution time is:

$$t_{parallel} = \left\lceil \frac{n}{P} \right\rceil t_{\#}$$

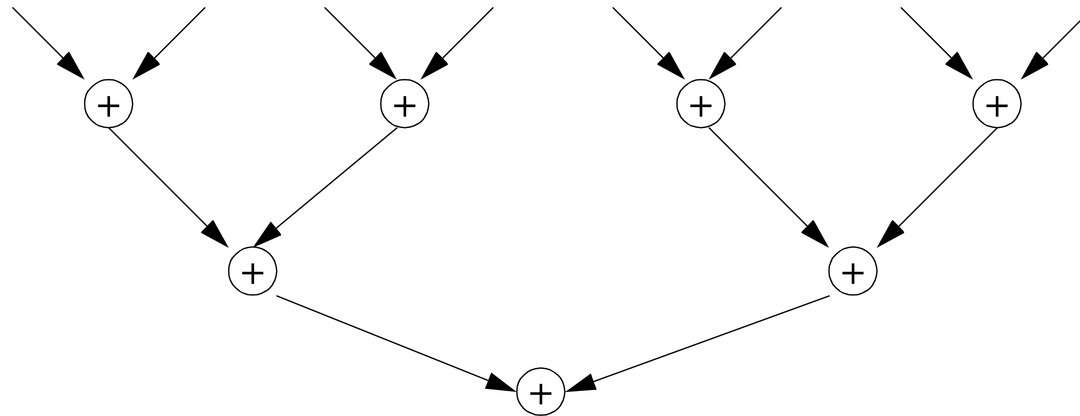
where $t_{\#}$ is the time to execute one # operation.

In a system where each processor contains an arithmentic pipeline, the execution time would be:

$$t_{parallel\&\text{pipeline}} = \left((s-1) + \left\lceil \frac{n}{P} \right\rceil \right) \tau$$

3 Time to Execute a Reduction

```
s = a(1) + a(2) + a(3) + ... a(n)
```



A sequence of $\lceil \log_2 n \rceil$ vector operations of length n/2, n/4, ..., 1 suffices to compute the reduction (assuming associativity).

Therefore (assuming $n=2^m$):

$$t_{pipeline} = \sum_{i=1}^{\log n} \left(s - 1 + \left(\frac{n}{2^i}\right)\right)\tau = ((s-1)\log n + (n-1))\tau$$

In the case of an array machine if the final reduction is done in logarithmic time, the execution time is:

$$t_{parallel} = \left\lceil \frac{n}{P} \right\rceil t_+ + \lceil \log P \rceil t_+$$

If $P >= n$, the time is:

$$t_{parallel} = \lceil \log n \rceil t_+$$

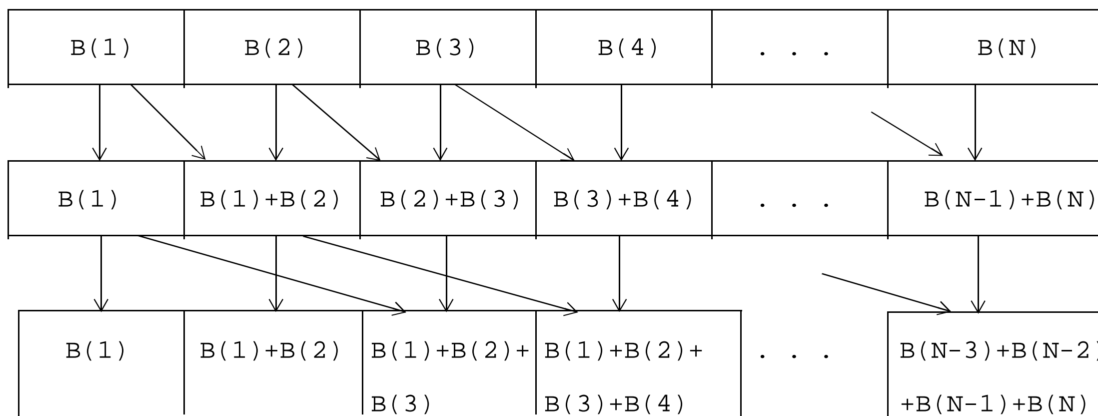
4 Parallel Prefix

Consider the following loop:

```
A(0)=0
DO I=1,N
   A(I)=A(I-1)+B(I)
END DO
```

The loop seems sequential because each iteration needs information on the value computed in the preceding iteration.

However, we can use a *parallel prefix* approach to compute the value of vector A in parallel as follows:



A parallel program implementing this strategy under the assumption that $N=2^k$ is:

```
A(1:N)=B(1:N)
DO I=1,K-1
   A(1+2**(I-1):N)=A(1+2**(I-1):N)+A(1:N-2**(I-1))
END DO
```

For an *array machine* with the number of procesing units *P>=n:*

$$t_{parallel} = t_{+}\lceil \log n \rceil$$

5 Examples of Speedup and Efficiency

Consider

```
a(1:n) + b(1:n)
```

The speedup, efficiency, and redundancy on a pipelined unit are:

$$S_s = \frac{s\tau n}{\tau[s + (n-1)]} = \frac{sn}{s + n - 1} \rightarrow s$$

$$E_s = \frac{s\tau n}{s\tau[s + (n-1)]} = \frac{n}{s + (n-1)} < 1$$

$$R_s = \frac{O_s}{O_1} = \frac{ns}{ns} = 1$$

In an array machine or multiprocessor:

$$t_{parallel} = \left\lceil \frac{n}{P} \right\rceil t_{+}$$

$$S_P = \frac{nt_{+}}{\left\lceil \frac{n}{P} \right\rceil t_{+}} = \frac{n}{\left\lceil \frac{n}{P} \right\rceil}$$

The value of S_P is P if n is a multiple of P.

$$E_P = \frac{nt_{+}}{P\left\lceil \frac{n}{P} \right\rceil t_{+}} = \frac{n}{P\left\lceil \frac{n}{P} \right\rceil}$$

E_P is 1 if n is a multiple of P. Otherwise it is < 1.

$$R_P = 1$$

The speedup, efficiency, and redundancy of the parallel prefix example on an array machine or multiprocessor with $P=n$ are:

$$S_n = \frac{nt_+}{\lceil \log n \rceil t_+} = \frac{n}{\lceil \log n \rceil}$$

$$E_n = \frac{nt_+}{n\lceil \log n \rceil t_+} = \frac{1}{\lceil \log n \rceil}$$

$$R_n = \frac{O_n}{O_1} = \frac{(n-1) + (n-2) + \ldots + \left(n - \frac{n}{2}\right)}{n} = \frac{n(\log n - 1) + 1}{n - 1} \approx \log n$$

6 Matrix-Vector Multiplication

In mathematical notation:

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \dots \\ V_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} A_{1i}V_i \\ \sum_{i=1}^{n} A_{2i}V_i \\ \dots \\ \sum_{i=1}^{n} A_{mi}V_i \end{bmatrix}$$

In Fortran:

```
do i=1,m
   R(i) = 0
   do j=1,n
      R(i) = R(i) + A(i,j) * V(j)
   end do
end do
```

```
do i=1,m
   R(i)=DOT_PRODUCT(A(i,1:n),V(1:n))
end do
```

The dot product is a vector multiplication (of length *n,* in this case) followed by a reduction.

Time in a pipelined machine for a dot product:

$$(s+n-1)\tau + ((s-1)\log n + (n-1))\tau = [(s-1)\log n + s + 2(n-1)]\tau$$

The total time for the matrix-vector multiplication is then:

$$m[(s-1)\log n + s + 2(n-1)]\tau$$

In an array machine or in a multiprocessor, the time if *P>n* is:

$$(t_{+} m(\lceil \log n \rceil + 1))$$

Alternatively, by interchanging the loop headers, the program could be written as follows:

```
do j=1,n
   do i=1,m
      R(i) = R(i) + A(i,j) * V(j)
   end do
end do
```

This leads to the following sequence of vector operations:

```
do j=1,n
   R(1:m)=R(1:m)+A(1:m,j)*V(j)
end do
```

The time for this loop in a pipelined machine is:

$$n(s+m-1)\tau$$

In an array machine or in a multiprocessor, the time (if P > m) is:

$$t_{+}2n$$

7 Matrix-Matrix Multiplication

1. *Inner product method.*

   Matrix multiplication is usually written:

```
do i=1,n
   do j=1,n
      do k=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
```

   The most direct translation of this program into vector form is:

```
do i=1,n
   do j=1,n
      C(i,j)=DOT_PRODUCT(A(i,1:n),B(1:n,j))
   end do
end do
```

The time on a pipelined machine is:

$$n^2((s-1)\log n + s + 2(n-1))\tau$$

The time on an array machine or multiprocessor if $P > n$ is:

$$(t_+\lceil \log n \rceil + t_*)n^2$$

2. *Middle-product method* (n-parallelism)

This is obtained by interchanging the headers in the original matrix multiplication loop.

```
do j=1,n
   do k=1,n
      do i=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
```

The direct translation of this loop into vector form is:

```
do j=1,n
   do k=1,n
      C(1:n,j)=C(1:n,j)+A(1:n,k)*B(k,j)
   end do
end do
```

Alternatively, the headers could have been exchanged in a different order to obtain the loop:

```
do j=1,n
   do k=1,n
      C(i,1:n)=C(i,1:n)+A(i,k)*B(k,1:n)
   end do
end do
```

The time on a pipelined machine is:

$$2n^2(s + (n-1))\tau$$

The time in an array machine is:

$$(t_+ + t_*)n^2$$

1. Outer-product method (n^2-parallelism)

Another interchange of the loop headers produce:

```
do k=1,n
   do i=1,n
      do j=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
```

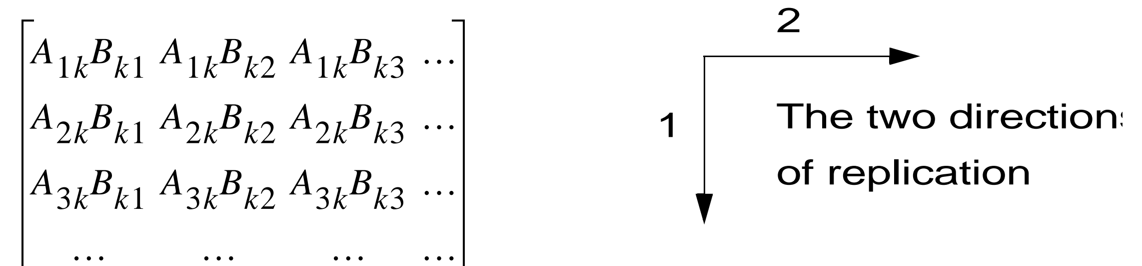
To obtain n^2 parallelism, the inner two loops should take the form of a matrix operation:

```
do k=1,n
   C(1:n,1:n)=C(1:n,1:n)+A(1:n,k)⊗ B(k,1:n)
end do
```

Where the operator ⊗ represents the *outer product* of two vectors. Given two vectors *a* and *b,* their outer product is a matrix *Z* such that $Z_{i,j}=a_i \times b_j$. Notice that the previous loop is

NOT a valid Fortran or Fortran 90 loop because ⊗ is not a valid Fortran character.

The outer product matrix in the loop above has the following form:



$$\begin{bmatrix} A_{1k}B_{k1} & A_{1k}B_{k2} & A_{1k}B_{k3} & \dots \\ A_{2k}B_{k1} & A_{2k}B_{k2} & A_{2k}B_{k3} & \dots \\ A_{3k}B_{k1} & A_{3k}B_{k2} & A_{3k}B_{k3} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

This matrix is the element-by-element product of the following two matrices:

$$\begin{bmatrix} A^k & A^k & \dots & A^k \end{bmatrix} \times \begin{bmatrix} \boldsymbol{B}^k \\ \boldsymbol{B}^k \\ \dots \\ \boldsymbol{B}^k \end{bmatrix}$$

which are formed by replicating A^k=`A(1:n,k)` and B^k=`B(k,1:n)` along the appropriate dimensions. This

replication can be achieved using the Fortran 90 `SPREAD` function discussed above:

```
spread(A(1:n,k),dim=2,ncopies=n)*spread(B(k,1:n),dim=1,ncopies=
n))
```

The resulting loop is therefore:

```
do k=1,n

C=C+SPREAD(A(1:n,k),2,n)*SPREAD(B(k,1:N,1,n)
end do
```

In an array machine with $P>n^2$, the time would be:

$$(2t_{\text{copy}}\lceil \log n \rceil + t_{*} + t_{+})n$$

where t_{copy} is the time to copy a vector. The time to `spread` to n copies is logarithmic as discussed in class.

2. n^3 parallelism

The product of two n×n matrices, `C=matmul(A,B)`, can be computed by adding n matrices of rank (n,n):

C =

$A_{11}B_{11}$ $A_{11}B_{12}$ $A_{11}B_{13}$ ...
$A_{21}B_{11}$ $A_{21}B_{12}$ $A_{21}B_{13}$ ...
...

\+

$A_{12}B_{21}$ $A_{12}B_{22}$ $A_{12}B_{23}$ ...
$A_{22}B_{21}$ $A_{22}B_{22}$ $A_{22}B_{23}$ ...
...

\+

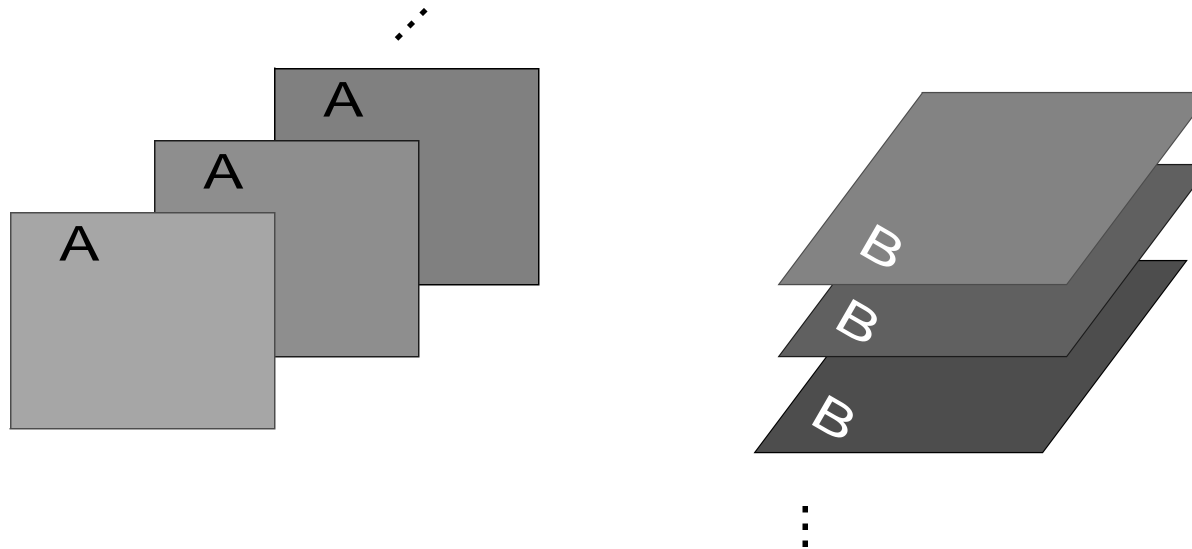
$A_{13}B_{31}$ $A_{13}B_{32}$ $A_{13}B_{33}$ ...
$A_{23}B_{31}$ $A_{23}B_{32}$ $A_{23}B_{33}$ ...
...

\+

...

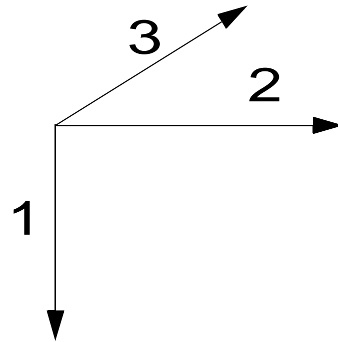
These n matrices of rank (n,n) can be computed by multiplying (element-by-element) two three-dimensional arrays of rank(n,n,n).

The two three-dimensional arrays are formed by replicating A and B along different dimensions as shown next:



This replication can, again, be achieved, with `SPREAD`.

Thus, give the following three directions of replication:



we can start by computing a n^3 temporary array T as follows:

```
T(:,:,:)=SPREAD(A,DIM=3,NCOPIES=n)*SPREAD(B,DIM=1,NCOPIES=n)
```

Then, the result is just `C=SUM(T,DIM=2)`

In an array machine with $P>=n^3$ processing unit, the time to compute C would be:

$$(2t_{\text{copy}}\lceil \log n \rceil + t_* + t_+\lceil \log n \rceil)$$

8 Multiplication by Diagonals

An $n \times n$ matrix A is banded if $A_{ij}=0$ for $i-j \geq \beta_1$, $j-i \geq \beta_2$:

$$
\begin{bmatrix}
A_{11} & A_{12} & \cdots & A_{1,\beta_2} & 0 & 0 & 0 \\
\cdots & A_{22} & A_{23} & \cdots & A_{2,\beta_2+1} & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
A_{\beta_1,1} & \cdots & \cdots & \cdots & \cdots & \cdots & A_{n-\beta_2+1,n} \\
0 & A_{\beta_1+1,2} & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \cdots & \cdots & \cdots & \cdots & A_{n-1,n} \\
0 & 0 & 0 & A_{n,n-\beta_1+1} & \cdots & \cdots & A_{n,n}
\end{bmatrix}
$$

For a small band, for example, $\beta_1=\beta_2=3$, the algorithm discussed before for matrix-vector multiplication is not efficient.

An alternative is to do the product by diagonals:

$$\begin{bmatrix} A_0 & A_1 & \dots & A_{\beta_2} & 0 & 0 & 0 \\ A_{-1} & \dots & \dots & \dots & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ A_{-\beta_1} & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \dots & \dots & \dots \end{bmatrix} \times V$$

After separating the diagonals into separate matrices, we get:

$$\begin{bmatrix} A_0 & 0 & 0 & 0 & 0 \\ 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots \end{bmatrix} V + \begin{bmatrix} 0 & A_1 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} V + \dots + \begin{bmatrix} 0 & 0 & 0 & A_{\beta_2} & 0 \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} V + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ A_{-1} & 0 & 0 & 0 & 0 \\ 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} V + \dots + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ A_{\beta_2} & 0 & 0 & 0 & 0 \\ 0 & \dots & 0 & 0 & 0 \end{bmatrix} V$$

which can be written as follows:

$$A_0 V \underset{+}{\wedge} A_1 V^2 \underset{+}{\wedge} \dots \underset{+}{\wedge} A_{\beta_2} V^{\beta_2} + A_{-1} V_{n-1} \overset{+}{\vee} \dots \overset{+}{\vee} A_{-\beta_1} V_{n-\beta_1}$$

where $V^j=(V_j,\dots,V_n)$ and $V_{n-j}=(V_1,\dots,V_{n-j})$.

Also, $\wedge$ means add the sorter vector to the first component of the longer one, and $\vee$ means add the shorter vector to the last component of the longer one.

In Fortran 90 (except for the greek letters and the subscripts):

```
     A_0(1:n)*V(1:n)  +
(/   A_1(1:n-1)*V(2:n),0. /)  +
...
(/   A_β1(1:n-β_1)*V(β_1+1:n), (0., j=1,β_1) /) +
(/   0., A_-1(1:n-1)*V(1:n-1) /) +
...
(/ (0., j=1,β_2), A_β2(1:n-β_2)*V(1:n-β_2) /)
```

9 Consistent Algorithms

A vector algorithm for solving a problem of size *n* is consistent with the best serial algorithms for the same problem if the redundancy is bounded as n $\rightarrow \infty$.

Vector reduction is consistent.

$$R_n = \frac{O_n}{O_1} = \frac{\frac{n}{2} + \frac{n}{4} + \ldots + 1}{n-1} = \frac{n-1}{n-1} \rightarrow 1$$

But parallel prefix is not (see Section 8.9).

The time for the simple form of parallel prefix presented in Section 8.7 given P processing units and vector length $n=2^k$ is:

$$t_{parallel} = \sum_{i=0}^{k-1} \left\lceil \frac{n-2^i}{P} \right\rceil$$

The problem with algorithms that are non consistant is that for n (size of input data) large enough, the array algorithm is

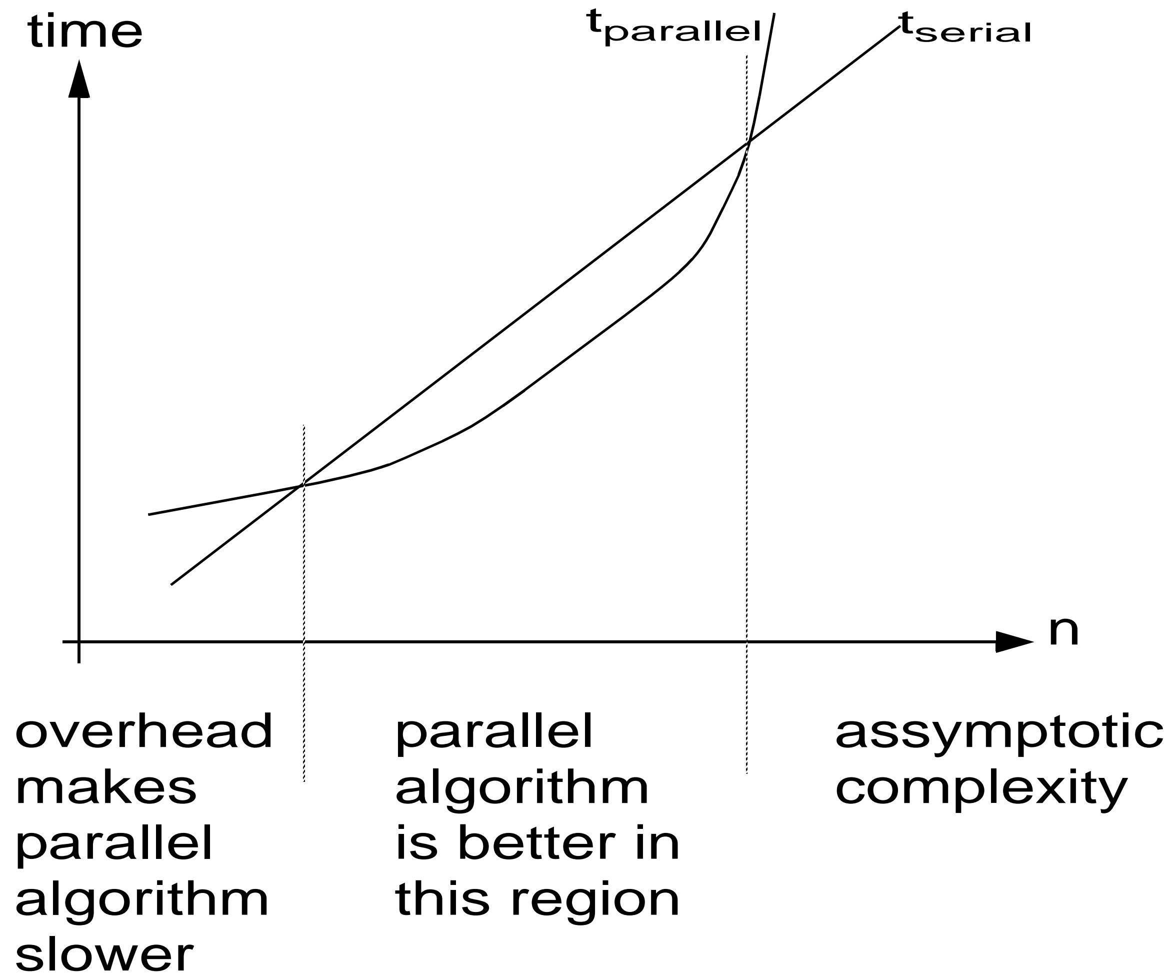
slower than the scalar version. Assuming constant number of devices (pipeline stages or processing units).

For example, assuming 8 processors $t_{parallel}$ has the following values:

n	t_{serial}	$t_{parallel}$
16	15	7
32	31	17
64	63	41
512	511	513
1024	1023	1153
$\sim 10^6$	$\sim 10^6$	$\sim 2.5 \times 10^6$

Notice that the array algorithm becomes slower than the scalar version when n=512.

A typical situation with inconsistent algorithms is depicted in the following graph:



10 A consistent version of parallel prefix

A consistent version of parallel prefix can be obtained by blocking the original algorithm. This can be done for both a pipelined unit or an array machine. We will show the array machine version. There are several steps in the algorithm

1. First the array B(1:n) should be reshaped into an $\left\lceil \frac{n}{P} \right\rceil \times P$ matrix as follows:

   $$\texttt{C=RESHAPE(B,(/}\ \left\lceil \frac{n}{P} \right\rceil \texttt{, P /),0.0)}$$

   The `reshape(source,shape[,pad][,order])` function takes the elements of `source`, in normal Fortran order, and returns them (as many as will fit) as an array whose shape is specified by the one-dimensional integer array `shape`. If there is space remaining, then `pad` must be specifie and is used to fill out the rest. See the manual for a description of `order`.

   Thus, if

```
B = (/1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11. /)
```

then, the result of `RESHAPE` above would be:

$$\begin{bmatrix} 1. & 5. & 9. \\ 2. & 6. & 10. \\ 3. & 7. & 11. \\ 4. & 8. & 0. \end{bmatrix}$$

2. Then, we assign a column of `C` to each processing unit and compute the partial sums of each column separately as follows:

```
do i=2, ⌈n/P⌉
   C(i,1:P) = C(i-1,1:P) + C(i,1:P)
end do
```

At the end of this loop we will have

$$C(k, q) = \sum_{i=1}^{k} C(i, q)$$

3. The third step is to compute the true value of the last element of each column:

```
do j=2,P
   C( ⌈n/P⌉ ,j)= C( ⌈n/P⌉ ,j-1)+C( ⌈n/P⌉ ,j)
end do
```

   At the end of this step, the last element of each column will have the sum of all elements in its own column and the columns preceding it.

4. Finally, the elements in each processor are adjusted:

```
do i=1, ⌈n/P⌉ -1
   C(i,2:P) = C( ⌈n/P⌉ ,1:P-1) + C(i,2:P)
end do
```

The total number of operation in the algorithm is:

$$\left(\left\lceil \frac{n}{P} \right\rceil - 1\right)P + (P-1) + \left(\left\lceil \frac{n}{p} \right\rceil - 1\right)(P-1)$$

The algorithms is consistent because:

$$\frac{\left(\left\lceil \frac{n}{P} \right\rceil - 1\right)P + (P-1) + \left(\left\lceil \frac{n}{p} \right\rceil - 1\right)(P-1)}{n} \to 2$$