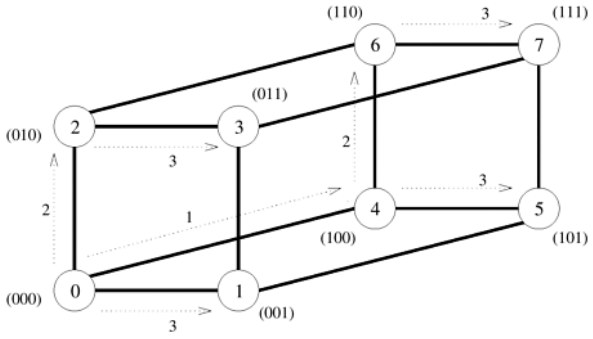
I. HYPERCUBE OF k PROCESSORS WITH ONE TO ALL BROADCAST (hypercube_broadcast.c)

Formulation:



The **entire array** generated at the source node (0) to every other process in $t_w \cdot m \cdot \log_2 k$ time of the entire array. The following is the sequence of communication (array transfers) between processes for each time step.

Iteration	Array Transfers	Word Transfers
1	(0 → 4)	n
2	(0 → 2), (4 → 6)	2n
3	(0 → 1), (2 → 3), (4 → 5), (6 → 7)	4n

The total number of word transfers equals $n + 2n + 4n + \ldots + 2^{\log_2 k} \cdot n = n \cdot (k-1) = O(nk)$. The following logic explains how the broadcast works. Initially, two variables
`mask` $= 2^{\log_2 k} - 1 =$ `111`
`check` $= 2^0 =$ `001`

At every iteration, the process with `(virtual_id & mask) == check` will receive data from the process with rank `id - check`. Once it has received the data, in the next iteration, it will send data to another process with rank `id + check`. After every iteration mask is right shifted by 1 bit (with 0 inserted) whereas check is left shifted by 1 bit (with 0 inserted).

After each of the processes (nodes) has the entire array, each process calculates the sum in $c \cdot n/k$ time. The sums are gathered in a sequence that is exactly the opposite to the broadcast operation. Thus, the total communication time here is $\log_2 k$.

Hence, the total time taken in this approach would be

$$T = \underbrace{t_w \cdot m \cdot \log_2 k}_{\text{broadcast time}} + \underbrace{t_w \cdot \log_2 k}_{\text{accumulate time}} + \underbrace{k \cdot (c \cdot n/k)}_{\text{summing time}}$$

MPI primitives used:

- `int MPI_Init( int *argc, char ***argv )`
- `int MPI_Finalize( void )`
- `int MPI_Comm_rank( MPI_Comm comm, int *rank )`
- `int MPI_Get_processor_name( char *name, int *resultlen )`
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,MPI_Comm comm)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Finalize( void )`

Parameter ranges experimented with:

Given the parameters n, p, k, Experimented with

- $n = 10^3$ to 10^9, in multiples of 10
- $k = 1$ to 6, in multiples of 2

Results:

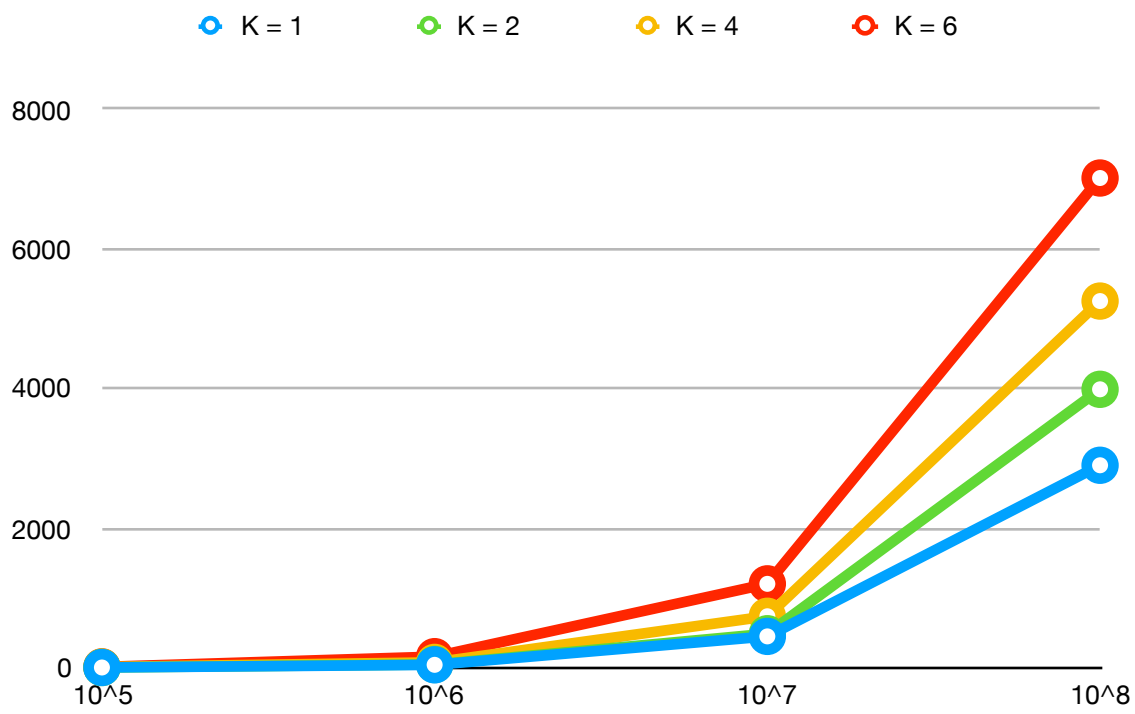


Figure 1: CPU time v/s array size

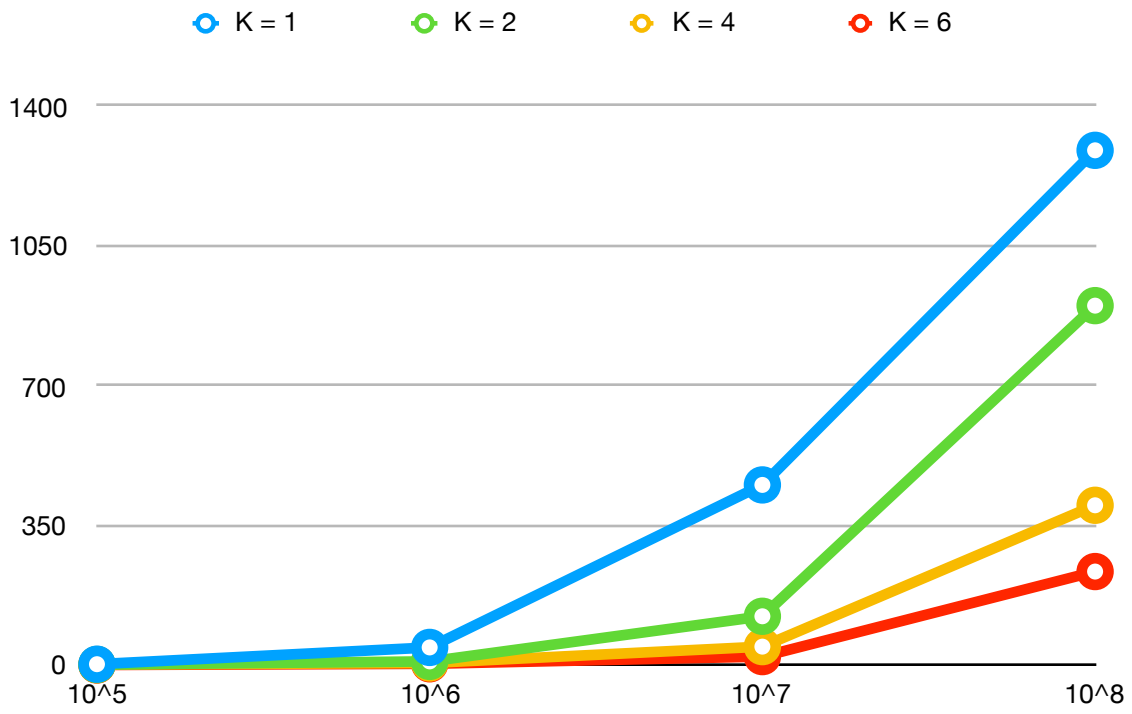


Figure 2: Wall time v/s array size

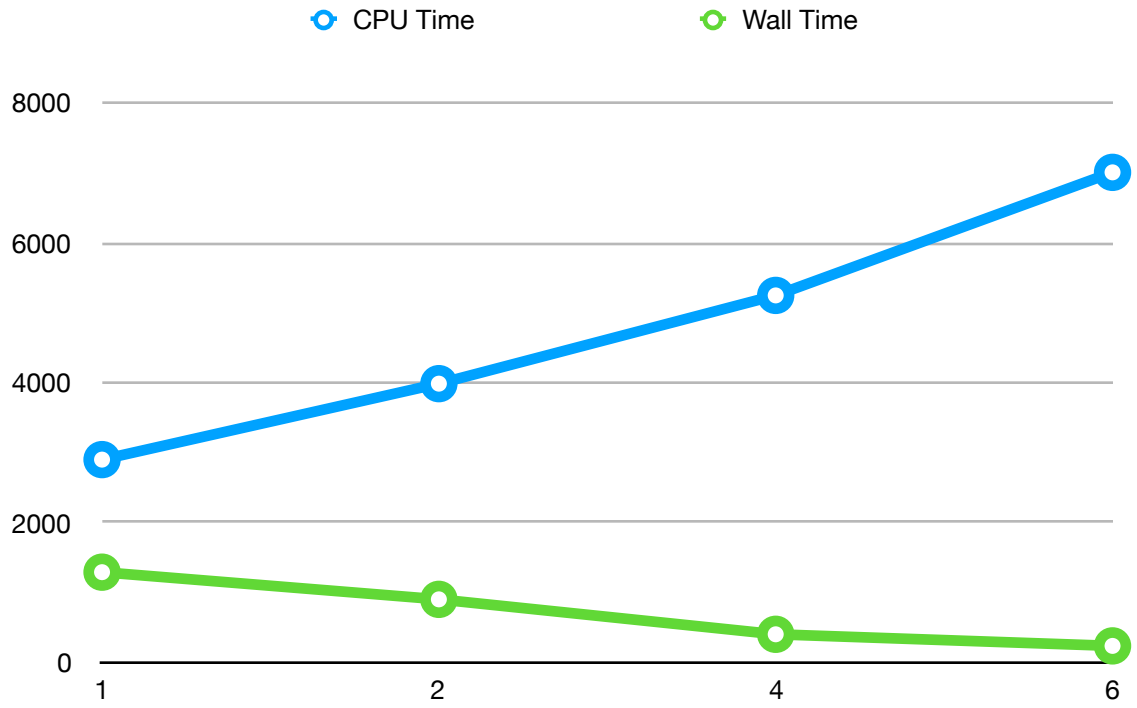


Figure 3: CPU Time and Wall time v/s k for $n = 10^8$

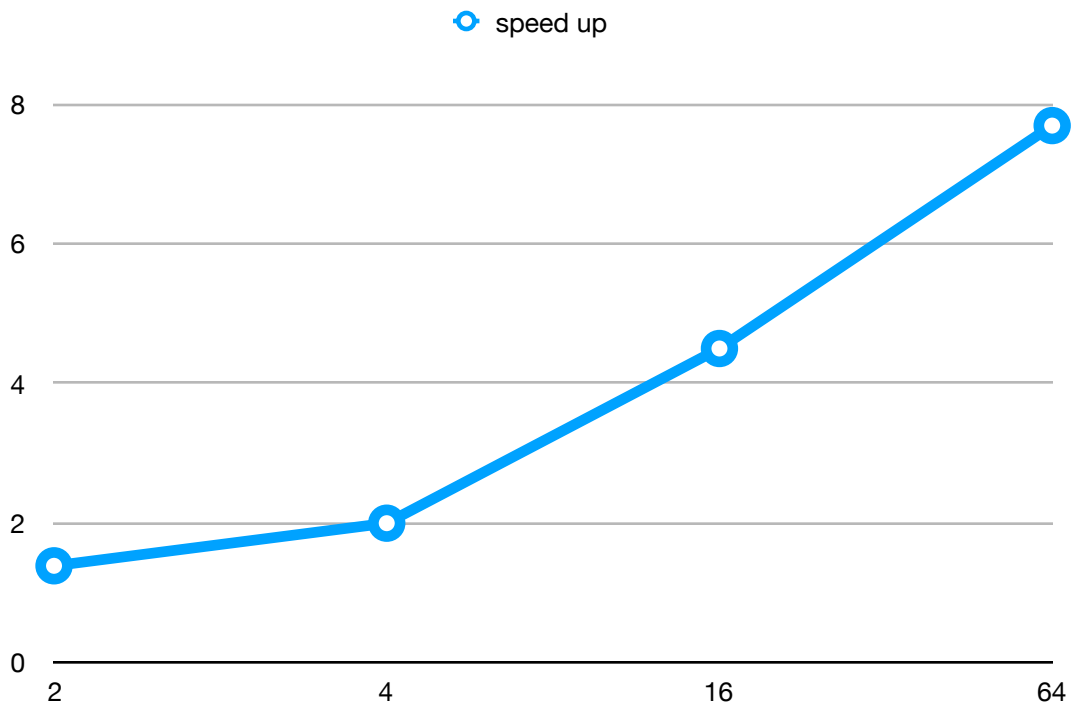


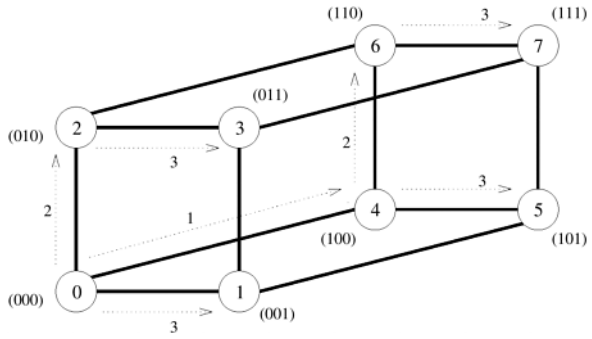
Figure 4: Speed up

Analysis:

- As the size of the array n increases, so does the CPU Time and Wall Time (as expected), as seen in **Figure 1** and **Figure 2**.
- Given a fixed value of n, the CPU Time **increases** while the Wall time **decreases** as k is increased as seen in **Figure 3**. This is expected since more computational and more importantly communication resources are used when the number of processes and processors is increased.
- From **Figure 3**, we can see that the Wall Time decreases as a rate lower than the CPU Time. Thus, this is somehow a representation of the efficiency of the parallel program, which is clearly visible in figure 4. Since the scale on the x-axis logarithmic and the speedup looks linear, the **speedup** is actually **sub-linear**, which is an indication of lower parallel efficiency. We might be able to increase the efficiency by reducing the amount of communication, which is what the **hypercube_scatter.c** program does.

I. HYPERCUBE OF k PROCESSORS WITH ONE TO ALL SCATTER (hypercube_scatter.c)

Formulation:



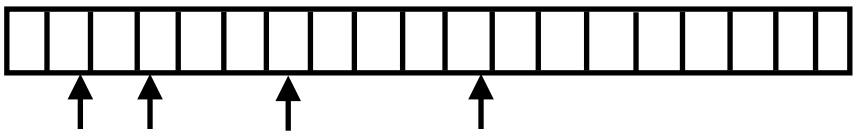
The **entire array** generated at the spruce node (0) to every other process in $t_w \cdot m \cdot \log_2 k$ time of the entire array. The following is the sequence of communication (array transfers) between processes for each time step.

Iteration	Array Transfers	Word Transfers
1	(0 → 4)	n
2	(0 → 2), (4 → 6)	$2 \cdot n/2$
3	(0 → 1), (2 → 3), (4 → 5), (6 → 7)	$4 \cdot n/4$

The total number of word transfers equals
$n + n + \ldots \log_2 k$ **times** $= n \cdot (\frac{(\log_2 k)^2 + \log_2 k}{2}) = O(n \cdot \log_2^2 k)$. The following logic explains how the broadcast works. Initially, the variables

`mask` $= 2^{\log_2 k} - 1 =$ `111`
`check` $= 2^0 =$ `001`
`num_to_distribute` $= n$

Before every iteration, the value of `num_to_distribute` is halved so that after during each iteration half the array (compared to the previous iteration) is transferred between two processors. Hence, the number of transfers in each iteration is n as seen in the table above and the the total transfers is logarithmic (instead of linear as in the first formulation).



array size transferred after each iteartion

At every iteration, the process with `(virtual_id & mask) == check` will receive data from the process with rank `id - check`.

Once it has received the data, in the next iteration, it will send data to another process with rank `id + check`. After every iteration mask is right shifted by 1 bit (with 0 inserted) whereas check is left shifted by 1 bit (with 0 inserted).

Hence, the total time taken in this approach would be

$$T = t_w \cdot m \cdot \log_2 k \ + \ t_w \cdot log_2 k \ + \ k \cdot (c \cdot n/k)$$

MPI primitives used:

- `int` **`MPI_Init`**`( int *argc, char ***argv )`
- `int` **`MPI_Finalize`**`( void )`
- `int` **`MPI_Comm_rank`**`( MPI_Comm comm, int *rank )`
- `int` **`MPI_Get_processor_name`**`( char *name, int *resultlen )`
- `int` **`MPI_Send`**`(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,MPI_Comm comm)`
- `int` **`MPI_Recv`**`(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int` **`MPI_Finalize`**`( void )`

Parameter ranges experimented with:

Given the parameters n, p, k, Experimented with

- $n = 10^3$ to 10^9, in multiples of 10
- $k = 1$ to 6, in multiples of 2

Results:

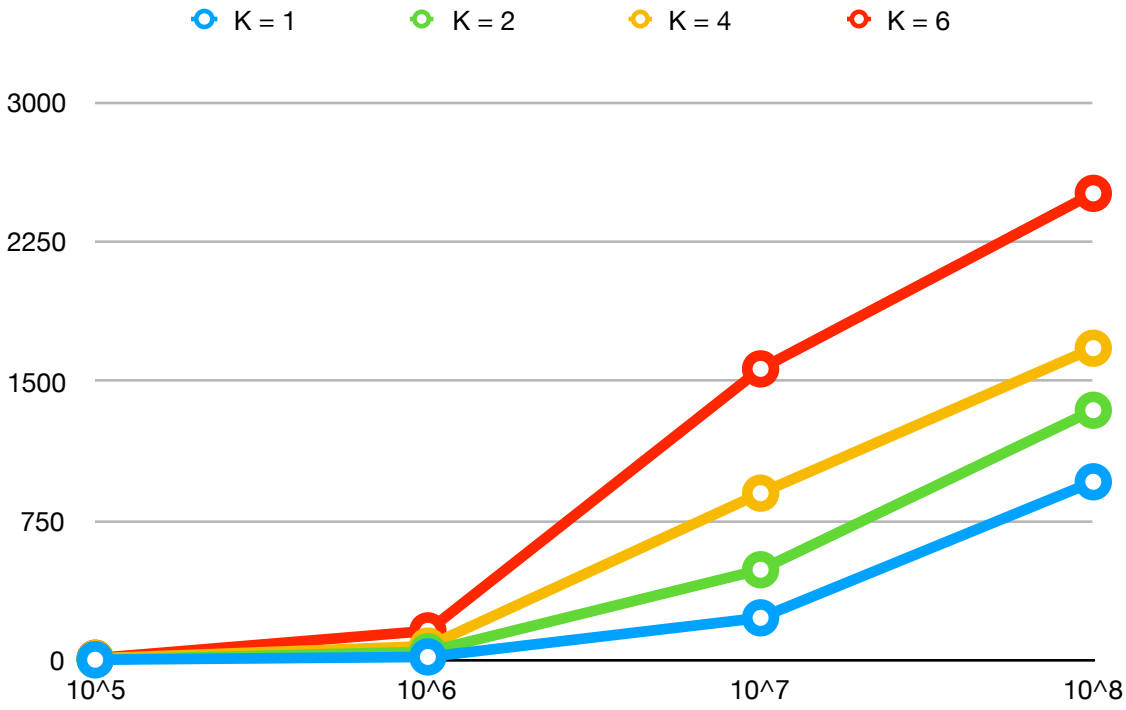


Figure 1: CPU time v/s array size

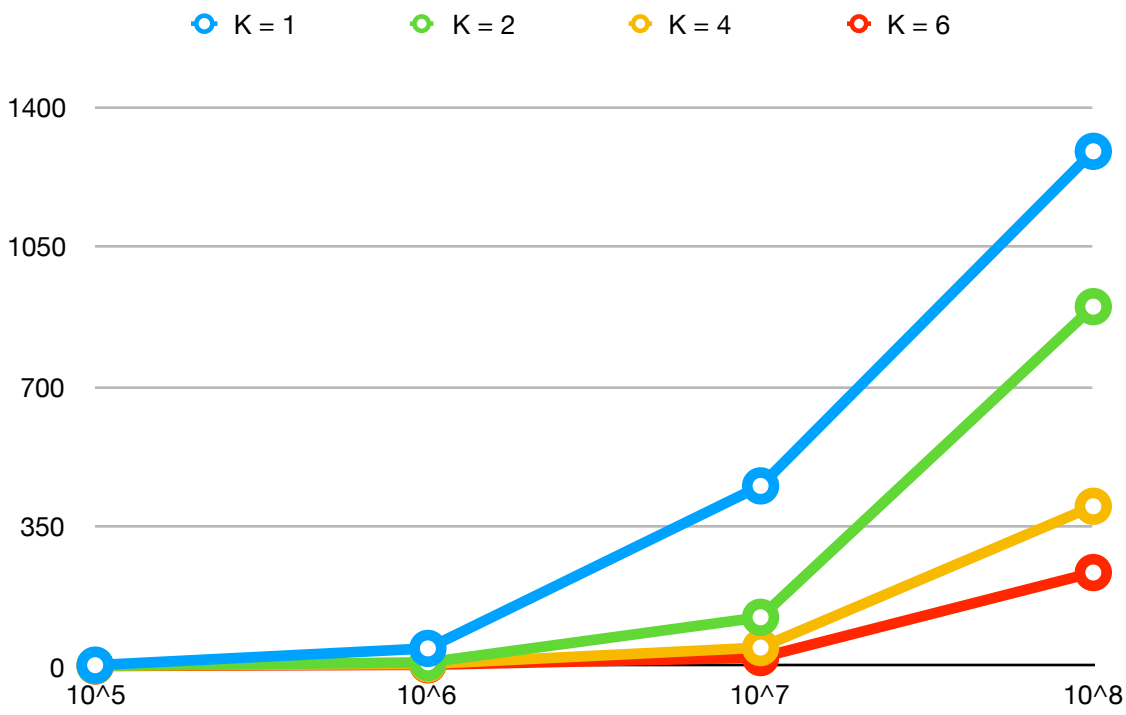


Figure 2: Wall time v/s array size

Analysis:

- As the size of the array n increases, so does the CPU Time and Wall Time (as expected), as seen in **Figure 1** and **Figure 2**.
- The wall time is almost similar to that in the first formulation.
- As seen from **Figure 1**, this time does not increase exponentially (especially for larger values of n). This is a result of reduction in the communication time due to $O(\log_2^2 k)$ running time of the scatter operation. Hence, it is clearly visible that the scatter operation has a huge advantage (and this advantage increases for larger values of n).
- This also means that the reduced efficiency in the first formulation was partially due to the communication overhead that dominated the computational time. Thus, using the scatter operation over the broadcast operation is a better method.