Lecture 6: Parallel Matrix Algorithms (part 1)

Matrix-Vector Multiplication

- Multiplying a dense $n \times n$ matrix A with an $n \times 1$ vector x: $y = Ax$

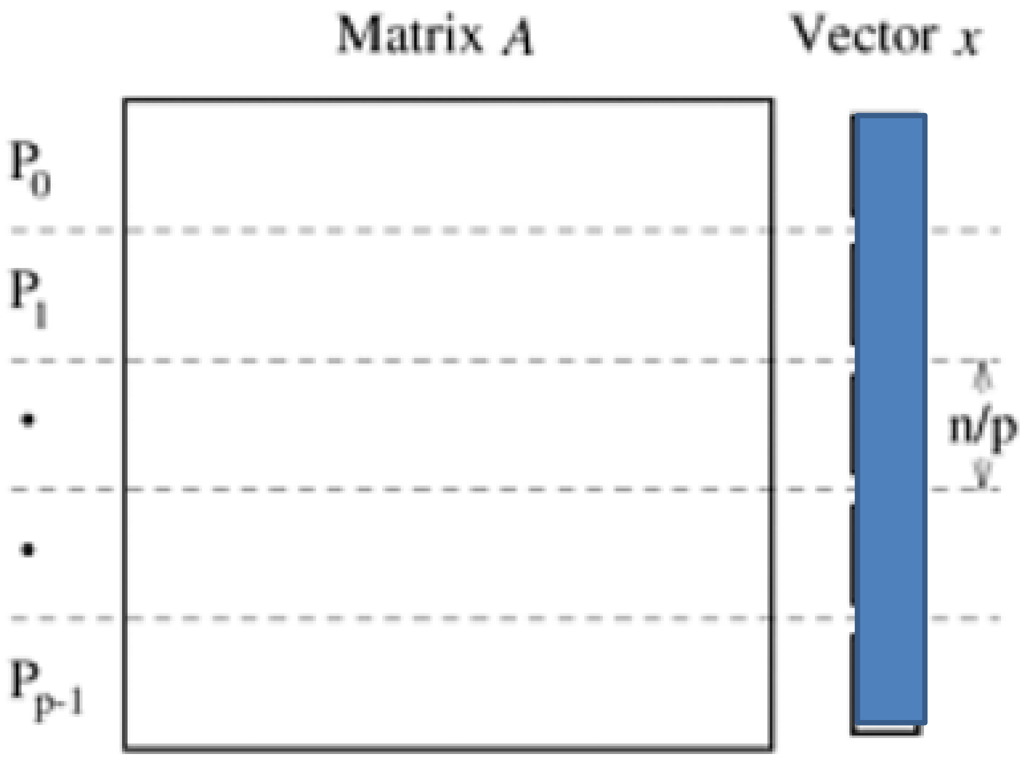
```
procedure MAT_VECT ( $A$, $x$, $y$)
begin
   for $i$ := 0 to $n$ - 1 do
   begin
      $y[i]$:=0;
      for $j$ := 0 to $n$ - 1 do
          $y[i]$ := $y[i] + A[i, j] \times x[j]$;
   endfor;
end MAT_VECT
```

- Sequential run time: $W = O(n^2)$

Row-wise Block-Striped Decomposition

Step 1

- Row-wise 1D block partition is used to distribute matrix.
- An all-to-all broadcast is used to distribute the full vector among all the processes.



Step 2

- Each task performs dot product computation using rows mapped to it and replicated vector x.

Step 3

- Distribute the result vector y to all processes by collective communication.
- MPI_Allgatherv(**void** **sendbuf*, **int** *sendcount*, **MPI_Datatype** *sendtype*, **void** **recvbuf*, **int** **recvcounts*, **int** **displs*, **MPI_Datatype** *recvtype*, **MPI_Comm** *comm*)

- MPI_Allgatherv(**void** **sendbuf*, **int** *sendcount*, **MPI_Datatype** *sendtype*, **void** **recvbuf*, **int** **recvcounts*, **int** **displs*, **MPI_Datatype** *recvtype*, **MPI_Comm** *comm*)
  - Gather data from all tasks and deliver the combined data to all tasks
  - The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf. These blocks need not all be the same size.
  - *recvcounts*: element j of this array is the number of elements being gathered from process j.
  - *displs*: element j of this array is the displacement from the first element of *recvbuf* where the first element gathered from process j is to be stored.

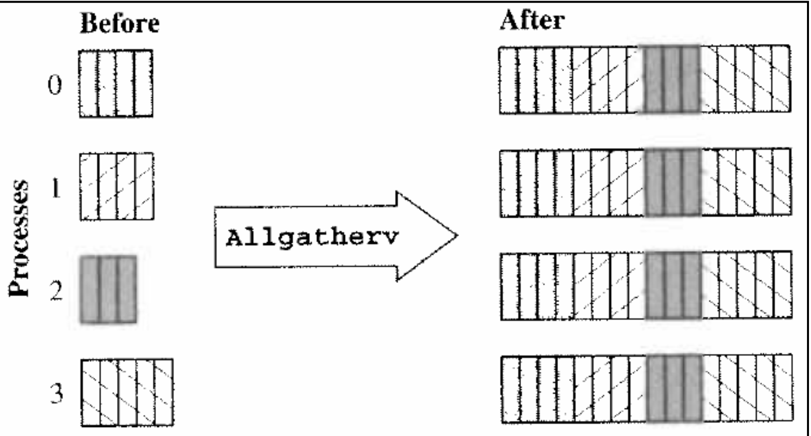


Figure 8.6 Function `MPI_Allgatherv` enables every process in a communicator to construct a concatenation of data items gathered from all of the processes in the communicator. If the same number of items is gathered from each process, the simpler function `MPI_Allgather` may be used.

```
void create_mixed_xfer_arrays(
        int     id,
        int     p,
        int     n,
        int     **count,
        int     **disp)
{
        int     i;
        *count = my_malloc(id, p*sizeof(int));
        *disp = my_malloc(id, p*sizeof(int));
        (*count)[0] = BLOCK_SIZE(0,p,n);
        (*disp)[0] = 0;

        for(i = 1; i < p; i++)
        {
            (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
            (*count)[i] = BLOCK_SIZE(i,p,n);
        }
}
```

This function creates the count and displacement arrays by scatter and gather functions, when the number of elements send/received to/from other processes varies

```
void replicate_block_vector(
        void        *ablock, /* block-distributed vector */
        int         n,
        void        *arep,   // replicated vector
        MPI_Datatype   dtype,
        MPI_Comm    comm)
{
        int         *cnt; // elements contributed by each process
        int         *disp;  // displacement in concatenated array
        int         id;
        int         p;

        MPI_Comm_size(comm, &p);
        MPI_Comm_rank(comm, &id);

        create_mixed_xfer_arrays(id, p, n, &cnt, &disp);
        MPI_Allgatherv(ablock, cnt[id], dtype, arep, cnt, disp, dtype, comm);
        free(cnt);
        free(disp);
}
```

replicate_block_vector() is used to transform a vector from a block distribution to a replicated distribution

Parallel Run Time Analysis

Message Passing Costs in Parallel Computers

- ***Startup time*** (t_s): The startup time is the time required to handle a message at the sending and receiving nodes. This include time to prepare the message(put in envelope), the time to execute routing algorithm, and the time to establish an interface between the local node and the router.
- ***Per-word transfer time*** (t_w): If the channel bandwidth is r words per second, then $t_w = \frac{1}{r}$.
- ***Per-hop time*** (t_h): After a message leaves a node, it takes a finite amount of time to reach the node in its next path.
- ***Cost model for communicating messages****:* Suppose that a message of size m is being transmitted through a network, Assume it traverses l links, the total communication cost is:

$$t_{comm} = t_s + lt_h + t_w m$$

- Assume that the # of processes p is less than n
- Assume that we run the program on a parallel machine adopting hypercube interconnection network (**Table 4.1** lists communication times of various communication schemes)

1. Each process is responsible for n/p rows of matrix. The complexity of the dot production portion of the parallel algorithm is $\Theta(n^2/p)$
2. Parallel communication time for all-to-all broadcast communication to replicate result vector y:
    - Each process needs to send a message of size n/p to all processes. This takes time $t_{comm} = t_s \log p + t_w \left(\frac{n}{p}\right)(p-1)$. Assume p is large, then $t_{comm} = t_s \log p + t_w n$.
3. The parallel run time for this program is:

$$T_p = \frac{n^2}{p} + t_s \log p + t_w n$$

4. This program is cost-optimal for $p = O(n)$

Remark: This analysis neglect the one-to-all communication to broadcast vector b initially.

Scalability Analysis

- Let $K = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$, then $T(n,1) = KT_0(n,p)$
- $T_0 = (t_s \log p + t_w n)p$
- Neglecting $t_w np$, $T(n,1) = Kt_s p \log p$. This is the isoefficiency with respect to message startup time.
- Neglecting $t_s p \log p$, $T(n,1) = Kt_w np$.

  Since $T(n,1) = n^2,\ T(n,1) = K^2 {t_w}^2 p^2$.

  In order to maintain a fixed efficiency, the problem size must increase with the rate of $\Theta(p^2)$.