

Programming a Distributed Matrix Transpose Using Mercury's Parallel Application System (PAS)

Application Note 205.0

TARGET AUDIENCE

Programmers familiar with general Parallel Application System (PAS) concepts interested in implementing a distributed matrix transpose for the RACE architecture or learning more about the PAS data movement functions.

ABSTRACT

This application note shows how to program a distributed matrix transpose for a RACE multicomputer using Mercury's Parallel Application System (PAS) library. Operations such as a distributed matrix transpose between arbitrary sets of processors demand a significant amount of "bookkeeping" to deal with the multitude of discrete data movements and the synchronization between processors. PAS encapsulates this data movement and inter-process synchronization with a single function call on each Compute Environment (CE). By taking advantage of Mercury's PAS library, programmers can develop high-performance multicomputing applications in significantly less time than previously possible.

Authored by:
Brian Bouzas, Systems Engineering

REVISION HISTORY		
Revision #	Date	Description
205.0	3/14/95	Creation

RACE and the RACE logo are registered trademarks of Mercury Computer Systems, Inc.

Mercury Computer Systems, Inc. believes this information sheet is accurate as of its publication date. Mercury Computers, Inc. is not responsible for any inadvertent errors. This information is subject to change without notice.

Copyright © 1995 Mercury Computer Systems, Inc.

Programming a Distributed Matrix Transpose Using Mercury's Parallel Application System (PAS)

This application note shows how to program a distributed matrix transpose for a RACE multicomputer using Mercury's Parallel Application System (PAS) library. This note assumes that the reader is familiar with general PAS concepts such as process sets and distributed buffers. For those unfamiliar with PAS, the PAS data sheet provides an introduction to basic PAS concepts.

Problem Statement

Assume that a matrix of $N \times M$ complex floating point elements is distributed over K Compute Environments (CEs) such that each CE has N/K rows of the full matrix with row elements stored sequentially in DRAM memory. In order to process complete columns of the matrix efficiently, we wish to redistribute the $N \times M$ matrix such that its transpose is distributed over a second set of L CEs. Each of the L CEs will then contain M/L transposed columns from the original $N \times M$ matrix where elements within a transposed column are stored sequentially in memory (Figure 1). For simplicity, assume that N is evenly divisible by K and M is evenly divisible by L , although PAS does not require even divisibility to perform the distributed transpose.

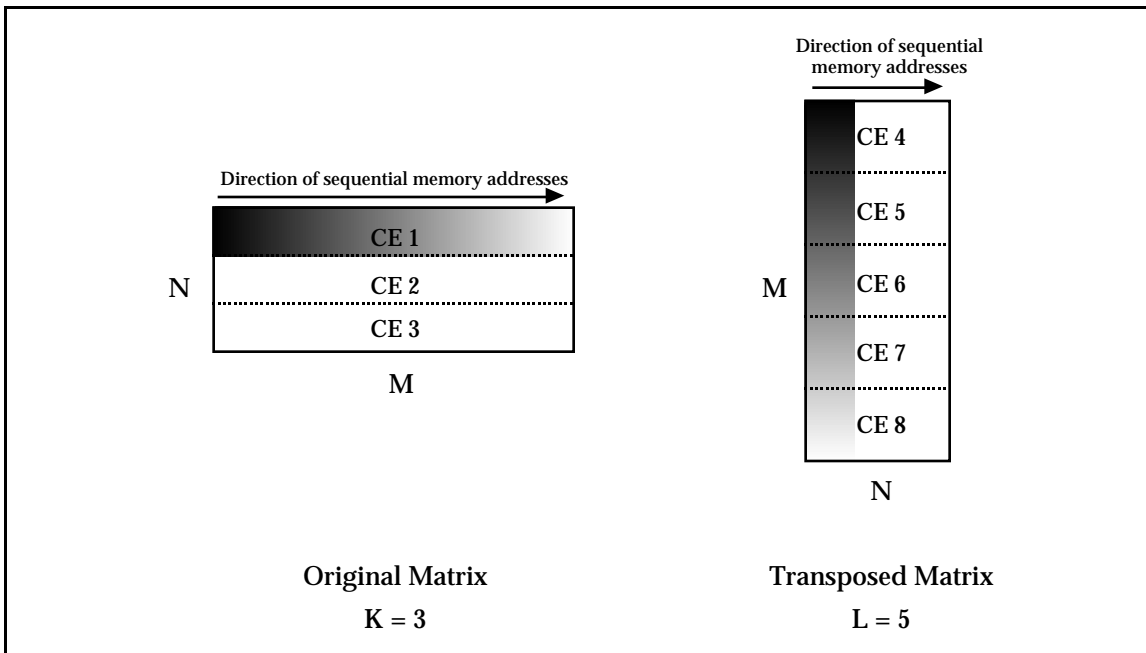


Figure 1. Distributed Matrix Transpose. An N -row by M -column matrix distributed by rows over K CEs is transposed and distributed by transposed columns over L CEs. The shaded region represents N/K rows of the original matrix.

To accomplish this operation, each of the K source CEs must orchestrate the transpose and transfer of M/L vectors of length N/K to the appropriate locations on each one of the L destination CEs. *Application Note 202* discusses the data movement associated with the distributed transpose in greater detail.

The `pas_movm()` Function

`pas_movm()`, "PAS move matrix", is the PAS function used to implement a distributed matrix transpose. The function provides general data movement capability for transferring matrix data between process sets.¹ Examples of `pas_movm()`'s capabilities include:

- Taking a matrix distributed in some manner (e.g. by bands of rows) across one process set and redistributing that matrix across another process set in a different manner (e.g. by bands of columns).
- Taking a complete matrix from one process and distributing it across a process set.
- Taking a complete matrix from one process and replicating it across a process set (i.e. broadcasting the matrix).
- Taking a matrix distributed across a process set and collecting it into a single process.

The distributed transpose described above is just a specific case of process-set-to-process-set data movement.

The prototype for `pas_movm()` is given here followed by a description of the key parameters:

```
typedef unsigned long ulong;

long pas_movm(
    PAS_id    active_procs,      /* Process set id for active processes */
    PAS_ptr   active_matrix,    /* PAS pointer to active process set matrix */
    ulong     active_tcols,     /* Zero, except for sub-matrix moves */
    long      active_attr,      /* Flags that describe active matrix */

    PAS_ptr   active_tmp_matrix, /* PAS pointer to temp matrix required with
                                PAS_TRANS_LOCAL flag */

    PAS_id    passive_procs,
    PAS_ptr   passive_matrix,
    ulong     passive_tcols,
    long      passive_attr,

    ulong     ncols,           /* Number of columns in the whole matrix */
    ulong     nrows,          /* Number of rows in the whole matrix */
    long      flag             /* Flag to describe data type, data movement,
                                and synchronization */
);
```

The active processes are the processes that perform the work needed to move the data. That is, the CPU or a DMA controller associated with each active process moves the data. The passive processes are those whose memories are written to or read from by the processes in the active process set. This implementation allows the active processes to be either the source or destination of the data transfer. If the source, the active processes write ("push") their data to the passive (destination) processes. If the destination, the active processes read ("pull") their data from the passive (source) processes. The transfer direction, "push or pull", is specified as part of the final `flag` argument. Note that `pas_movm()` does not require the active and passive process sets to be disjoint; it readily accommodates active and passive process sets that are identical or that overlap.

The attribute flags, `active_attr` and `passive_attr`, describe how the active and passive matrices are partitioned across their respective process sets (or single processes). For example, the original matrix in Figure 1 is said to be partitioned as "blocks of rows". Figure 2

¹The function `pas_movv()` provides analogous capabilities for vectors.

illustrates the distributions described by different combinations of attribute flags. In the case of the `PAS_BLOCK` attribute flag, the dashed lines in Figure 2 indicate the partitioning of the matrix into blocks. The number inside each block represents a PAS process number; typically, each PAS process runs on a different CE.

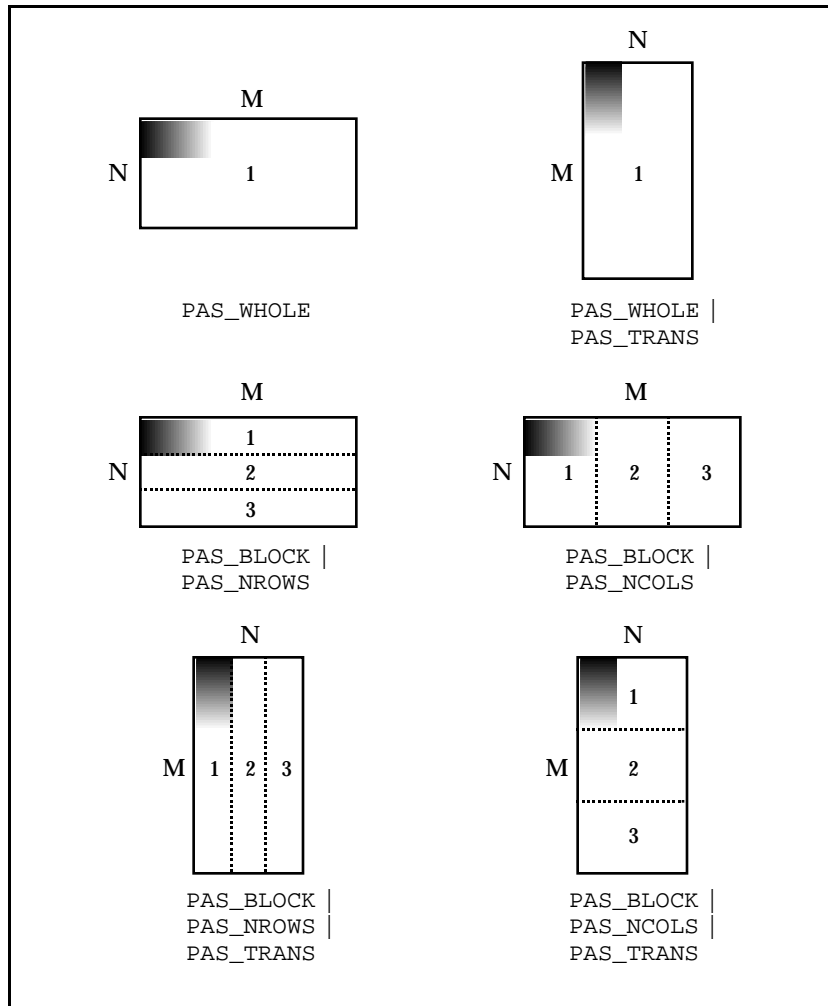


Figure 2. PAS Matrix Attribute Flags. Matrix distributions described by different combinations of PAS attribute flags for an N-row by M-column matrix. The shaded region represents $N/3$ rows and $M/3$ columns of the whole matrix.

As shown in the next section, transferring a matrix from one process set to another while simultaneously changing the way the data is organized is largely a matter of supplying the correct attribute flags to `pas_movm()` to describe the active and passive matrices.

In addition to specifying the transfer direction, the final `flag` argument to `pas_movm()` also specifies the data type (real or complex; float, long, or integer) and the transfer "engine" (a DMA controller or the CPU).

Programming the Transpose

The following example shows how to implement a distributed matrix transpose for the case where an $N \times M$ source matrix is partitioned by rows across the active process set and DMA controllers are used to move the data to the passive process set. Many variations of distributed matrix transpose are possible using `pas_movm()`; this example illustrates just one particular case.

Active Process Set (Source CEs)

To perform a distributed transpose, `pas_movm()` is called by each member of the active process set. In this example the active processes provide the source data. Therefore, one active process should be run on each of the K source CEs. The passive process set should span the L destination CEs. Since the passive processes do not move data, they do not call `pas_movm()`. Each process in the passive process set needs only to call `pas_sem_take()`, a synchronization function, to learn when its data has arrived.

Given the following PAS information,

<code>source_pset</code>	is the process set id for the PAS processes on the K source CEs
<code>dest_pset</code>	is the process set id for the PAS processes on the L destination CEs
<code>source_pp</code>	is a PAS pointer for the source matrix distributed by rows over the K source CEs
<code>dest_pp</code>	is a PAS pointer for the destination matrix distributed by transposed columns over the L destination CEs
<code>temp_pp</code>	is a PAS pointer for a scratch matrix on the K source CEs that may be used by <code>pas_movm()</code>

the following `pas_movm()` call on the K source CEs will produce the desired data movement:

```
pas_movm(
    source_pset,          /* Active process set id */
    source_pp,           /* Pointer to active matrix */
    0,
    PAS_BLOCK | PAS_NROWS, /* Active matrix attributes (see Figure 2) */
    temp_pp,            /* Pointer to scratch matrix */
    dest_pset,          /* Passive process set id */
    dest_pp,           /* Pointer to passive matrix */
    0,
    PAS_BLOCK | PAS_NCOLS | /* Passive matrix attributes (see Figure 2) */
    PAS_TRANS,
    M,                  /* Number of columns in whole matrix */
    N,                  /* Number of rows in whole matrix */
    PAS_COMPLEX | PAS_FLOAT | /* Data type */
    PAS_PUSH |         /* Direction of data movement */
    PAS_TRANS_LOCAL | PAS_DMA | /* Transpose method (see text) */
    PAS_SEM_GIVE_AFTER /* Give semaphore to each passive process after
                        accessing memory (see text) */
);
```

The attribute flags describe the active and passive matrices with respect to the `ncols` and `nrows` dimensions. For the active (source) matrix, `PAS_BLOCK` indicates that the matrix is

distributed across the process set; `PAS_NROWS` indicates that each active process contains some number of rows. Similarly, for the passive (destination) matrix, `PAS_BLOCK` indicates that the matrix is distributed across the process set; `PAS_NCOLS` indicates that each passive process contains some number of columns. The `PAS_TRANS` flag indicates that the passive matrix is transposed. That is, elements within a column of the original matrix are stored sequentially in the passive processes' memory.²

In the final set of flags, `PAS_COMPLEX` and `PAS_FLOAT` describe the data type. `PAS_PUSH` instructs the active (source) processes to write data to the passive (destination) processes. The `PAS_TRANS_LOCAL` flag tells the `K` active (source) CPUs to transpose their local part of the distributed matrix into the local scratch buffer at `temp_pp`. And the `PAS_DMA` flag causes a local DMA controller to move the transposed pieces from the scratch buffer to the appropriate locations in the `L` passive (destination) CEs.

Alternatively, the active CPUs can transpose and move their data directly to the destination processes in a single step without using a scratch buffer (i.e., `active_tmp_matrix`). With this approach, small blocks of the local buffer are read into on-chip memory by the active CPU, transposed, and written directly to the appropriate destination buffer. To perform the distributed transpose in this manner, the `PAS_TRANS_LOCAL | PAS_DMA` flags are replaced by `PAS_TRANS_DIRECT | PAS_CPU`. The `PAS` pointer to the temporary buffer can then be supplied as a `NULL` pointer.

Whether the direct or local transpose method achieves higher performance is a function of the matrix size, the number of source and destination CEs, as well as the underlying hardware topology. Because the analysis is complicated, determining the best transpose method for a particular application is usually easier done empirically.

In general, when the active processes are the source of a data movement orchestrated by `pas_movm()`, the passive (destination) processes must be told that their data has arrived. The `PAS_SEM_GIVE_AFTER` flag causes each member of the active process set to give a semaphore to each member of the passive process set when it has finished writing data to the passive process' memory. Members of the passive process set, therefore, know that all their data has arrived after they receive a semaphore from each member of the active process set.

Passive Process Set (Destination CEs)

A single call to `pas_sem_take()` tells a passive process to wait until it has received all of its data from the active processes; the passive process can then begin processing its piece of the transposed matrix. For the local transpose case shown above, the function call for the passive processes is `pas_sem_take(source_pset, PAS_DMA)`; if the CPUs are used to transpose and move the data directly, `PAS_DMA` should be replaced by `PAS_CPU`. The `pas_sem_take()` function will not return until a semaphore has been given by each member of `source_pset`.

Conclusion

The `pas_movm()` data movement function allows complex data transfers to be programmed with minimal effort. Operations such as a distributed matrix transpose between arbitrary sets of processors demand a significant amount of "bookkeeping" to deal with the multitude of discrete data movements and the synchronization between processors. `PAS` encapsulates this data movement and inter-process synchronization with a single function call on each CE -- `pas_movm()` for the active process set, `pas_sem_take()` for the passive process set. Without a multiprocessing library like `PAS`, the programmer must manage these bookkeeping details explicitly. By taking advantage of `PAS`, programmers can develop high-performance multicomputing applications in significantly less time than previously possible.

²By convention, row elements are stored sequentially in memory by default.