

PvmJobs: A Generic Parallel Jobs Library for PVM

Hong H. Ong Iyad A. Ajwa Paul S. Wang*
Institute for Computational Mathematics
Department of Mathematics & Computer Science
Kent State University
Kent, Ohio 44242, U.S.A.
Telephone: (330) 672-4004 ext. 111
Fax: (330) 672-7824
{hong , iajwa , pwang}@mcs.kent.edu

Abstract

PvmJobs is a general bag-of-jobs library for PVM that works with any user created job structure in a master/slave paradigm. A master can spawn slave processes, schedule and dispatch jobs to slaves, coordinate and synchronize the activities. A slave process obtains a job from the master, performs a set of prescribed tasks, returns results to the master, and obtains the next job. Slaves are organized into separate disjoint groups, called *bags*. Each bag has one master, its own set of slaves, and jobs to perform. A master may use one or more bags simultaneously, and a slave can be a master as well. *PvmJobs* provides a simple FIFO job scheduling mechanism which can be easily replaced by application-defined priority-driven scheduling. The package is written in C and is easy to use by anyone who knows PVM. *PvmJobs* is well documented and should be of interest to any PVM application that uses a master/slave message-passing paradigm. The package has been used in various typical parallel computations and applied in the parallel implementation of the Gröbner Bases Algorithm and the Characteristic Sets Method. The design and implementation of *PvmJobs* is presented. The library routines and their usage are described. Examples are given. General requirements of the library from applications are carefully explained.

1 Introduction

Parallel Virtual Machine (PVM) is a software facility that enables users to use a set of networked computers as one large parallel virtual machine. Hence, it provides an environment where users can exploit distributed/parallel computation across a wide range of computer types in an efficient and convenient manner. PVM has been widely used both in research and commercial areas. Its simplicity and robustness have encouraged the development of many parallel/distributed computing applications. Here at ICM/Kent, our research on parallel and distributed computations is made easier and more flexible by using PVM as a tool for communicating data between processing elements and for controlling parallelism. *PvmJobs*

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650

is part of the overall effort at ICM/Kent to build tools to integrate heterogeneous scientific compute engines.

PvmJobs is a general bag of jobs library that works with any user-defined job data structure in a master/slave paradigm. A master process is generally the control process whose main responsibilities are to spawn slave processes, perform initialization, monitor the status of all slave processes, schedule and distributes jobs to slave processes, and synchronize activities between slave processes and itself. On the other hand, a slave process will accept jobs from the master process and perform a specific task. Once the slave process completed the task, it returns the result to the master process and waits for the next job. In *PvmJobs*, slave processes are organized into separate and disjoint groups, called *bags*. Each *bag* has its own set of slave processes and jobs to perform. A master process may use one or more *bags* simultaneously. The slave processes in each *bag* will do the actual computation or may become a sub-master and distribute the workload to slave processes by creating more *bags*. This is illustrated in Figure 1.

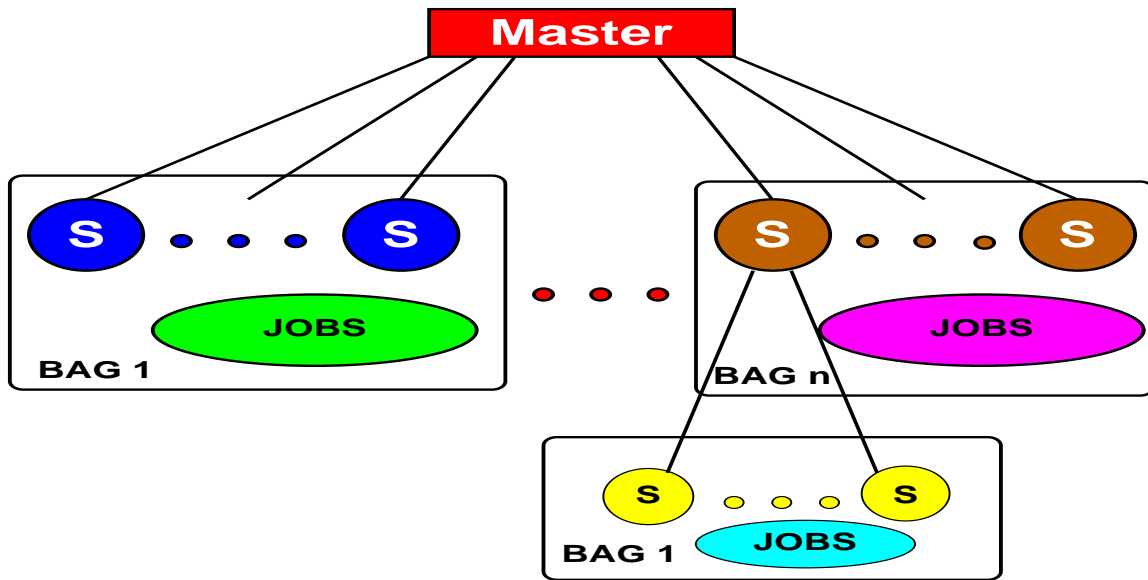


Figure 1: PvmJobs Library Structure.

A master process can use appropriate application-dependent scheduling algorithms for load balancing. An application can use any well-developed scheduling technique to meet its needs. A simple first-in-first-out (FIFO) scheduling is built into *PvmJobs*. However, the FIFO scheme can be easily replaced by a user-defined scheduling scheme.

We begin (in Section 2) by describing the design and implementation of *PvmJobs*. Section 3 presents the *PvmJobs* library application program interface (API) and usage. Section 4 gives an example that uses *PvmJobs* and shows what an application must do to use this tool. Section 5 summarizes the *PvmJobs* library.

2 Design And Implementation

The goal of *PvmJobs* is to provide a tool for PVM applications to employ the master-slave paradigm simply and easily. Thus, *PvmJobs* must be designed as a general purpose facility. The mechanism must be powerful, easy to understand, simple to use, and flexible enough to suit all types of applications. Our design is for *PvmJobs* to be a combination of library-supplied and application-supplied mechanisms. Thus, central design considerations are

1. What mechanisms should be supplied by the PvmJobs library?
2. What facilities should be supplied by the application program itself?
3. How are these two mechanisms interfaced?

To work with PVM, *PvmJobs* is implemented as a C library. For ease of use, high-level master and slave operations are provided that hide many details.

2.1 High-level Operations

For ease of use, *PvmJobs* defines high-level operations for the tasks that an application using the master-slave paradigm needs to perform. We discuss these operations next.

- **Initialization.** The master part of the application program will need to initialize certain data structures, determine the number of slave processes needed, and spawn the required processes. The master may also pass a set of initial data to the spawned slaves. Also, a master frequently needs to spawn slave processes to perform different duties. In *PvmJobs*, a *bag of jobs* contains a number of identical slaves and a set of jobs that can be farmed out to these slaves. Each bag of jobs may handle a specific computation distinct from other bags. Multiple bags can cooperate to solve a larger problem, or perform unrelated computations.
- **Scheduling and dispatching jobs.** After initialization, the master process needs to divide the jobs to be performed among available slaves and dispatch them to slave processes. Scheduling is done well if all slaves do more or less the same amount of work. The scheduling policy is application dependent and *PvmJobs* works with application-supplied scheduling policy. It allows applications to define their own scheduling scheme. The library provides a routine to dispatch a specific job to a specific slave process.
- **Requesting jobs and sending completed results.** Once a slave process is active, it will receive some initial data such as which *bag* it belongs to and/or arguments that it may need to perform it's own initialization from the master process. The library provides routines for a slave process to send results back to the master process and to request new jobs from that master.
- **Synchronization.** Sequencing and ordering the parallel activities between master and slave processes is very important. *PvmJobs* provides primitives for handling synchronization.
- **Receiving completed jobs.** The master process will receive a completed job/result from a slave process in a specific *bag* through a library call.

- **Termination.** The master process will notify slave processes when it is ready to terminate execution. This usually causes the slaves to terminate themselves as well.

2.2 Implementation

The *PvmJobs* library is implemented in two parts. The first part is composed of internal functions which maintain several data structures that are required by the *PvmJobs* library to maintain consistency between the *bags*. The internal functions also record the states of the library routines, the status of slave processes, and slave processes' readiness. In addition, a set of error constants and message tags are defined to facilitate users to identify errors returned from function calls and messages between processes.

The second part of the library consists of the API (application program interface) routines. API contains twelve functionally complete subroutines and is designed to be very general and transparent to the user. Detailed descriptions and usage of API routines are given in Section 3.

3 PvmJobs Application Programming Interface And Usage

In this section, we discuss routines of *PvmJobs* API interface. We present a detailed description of each API subroutine in Section 3.1. We discuss usage and a typical scenario of using the API in Section 3.2.

3.1 Library Interface

The library API consists of twelve user level functions which provide great flexibility of using *PvmJobs* library. We present these functions in the following subsections. But first, we give a general description of these functions.

The library routines are implemented using the general conventions used by most C systems. To be more specific, each function arguments are a combination of value parameters and pointers where appropriate; and function will return an integer value to indicate the outcome of the call. Application programs written in C access *PvmJobs* library routines by linking against an archival library (libpvmjobs.a) that is provided with the package.

3.1.1 bag_open

Open a new bag of jobs.

SYNOPSIS

```
int bd = bag_open( char *taskname, char **argv,
                  char **hosts, int nhosts, int ntasks);
```

PARAMETERS

```
taskname  Character string which is the executable file name
          of the slave process to be started.
  argv    Pointer to an array of arguments for the executable.
  hosts   Pointer to a string specifying where to start a PVM process
  nhosts  Number of hosts in "hosts".
  ntasks  Number of copies of the executable to start.
```

bd Integer returning the bag descriptor.

DISCUSSION

The `bag_open` routine starts `ntasks` copies of the executable named `taskname` on hosts given in the `hosts` argument. `bag_open` uses round-robin assignment to distribute the `ntask` slave processes across the virtual machine specified by the `hosts` argument. On return, `bd` contains the bag descriptor for the bag of jobs started. The bag descriptor is an integer value for the master program to differentiate opened bags and identify which bag that the slave processes are in. Negative bag descriptors indicate errors.

3.1.2 bag_close

Tell the *PvmJobs* library that application program is closing the specified bag.

SYNOPSIS

```
int cc = bag_close( const int bd )
```

PARAMETERS

bd Bag descriptor. An integer identifying a bag.

DISCUSSION

The `bag_close` routine tells the *PvmJobs* library that application program is closing the bag indicated by `bd`. `bag_close` will terminate all slave processes in the specified bag and claim back the bag descriptor.

3.1.3 bag_complete

Received a completed job from any slave process that belong to the specified bag.

SYNOPSIS

```
int cc = bag_complete(const int bd, int *tid, ANS answer  
                    UNPACK_ANS_FN unpack_ans)
```

PARAMETERS

bd Bag descriptor. An integer assigned by *PvmJobs* library.

tid Task id of the slave that returned a completed job.

answer Pointer to an answer structure.

unpack_ans Pointer to function which will unpack the completed job.

cc Status code returned by the routine.

DISCUSSION

The `bag_complete` routine is used by the master process to receive a completed job from any slave process. Also, it is used to accept a request for a new job from a slave process. The argument `bd` is used to determine which bag's slave processes should `bag_complete` listen to and receive a completed job from. The application program must also define a function "unpack_ans" to unpack the completed job returned by the slave process.

3.1.4 bag_dispatch

Dispatch a job to a ready slave process.

SYNOPSIS

```
int cc = bag_dispatch(const int bd, const int tid,  
                    JOB job, PACK_JOB_FN pack_job)
```

PARAMETERS

bd Bag descriptor. An integer assigned by PvmJobs library
tid Task identifier of a PVM process.
job Pointer to a job structure.
pack_job Pointer to user's defined packing function to pack a job.
cc Status code returned by the routine.

DISCUSSION

The `bag_dispatch` routine is used by the master process to pack and send a job to a ready slave process identified by `tid`. If `tid` is not specified, `bag_dispatch` will select a slave process from its own ready queue according to a (FIFO) scheme. `bag_dispatch` routine is asynchronous. Once the job is packed and delivered, `bag_dispatch` passes control back to the application program.

3.1.5 bag_displaytasks

Display the PVM task identifier of slave processes in the specified bag.

SYNOPSIS

```
int status = bag_displaytasks( const int bd )
```

PARAMETERS

bd Bag descriptor. An integer assigned by PvmJobs library
status Integer status code returned by the routine.

DISCUSSION

The `bag_displaytasks` routine is called by master process to display the tasks `tid` of the PVM processes identified by `bd`. It also prints out the error code of failed PVM processes.

3.1.6 bag_perror

Print message describing the last error returned by a *PvmJobs* library call.

SYNOPSIS

```
int info = bag_perror( const char *sb )
```

PARAMETERS

sb Character string supplied by the user which will be prepended to the error message of the last PvmJobs library call.
info Integer status code returned by the routine.

DISCUSSION

The `bag_perror` routine returns the error message of the last *PvmJobs* library call. The user can use `sb` to add additional information to the error message, for example, its location.

3.1.7 bag_gettaskid

Returns an array of task identifiers in the specified bag.

SYNOPSIS

```
int numt = bag_gettaskid( const int bd, const int ntasks, int *tids)
```

PARAMETERS

bd Bag descriptor. An integer assigned by PvmJobs library.

ntasks Number of task identifiers to return.
tids Array of task identifiers.
numt Actual number of active processes started.

DISCUSSION

The `bag_gettaskid` routine returns `ntasks` task identifiers from bag `bd`. The argument `numt` contains the actual number of active tasks.

3.1.8 bag_addtasks

Add more slave processes to the specified bag.

SYNOPSIS

```
int numt = bag_addtasks(const int bd, char **argv,  
                        char **hosts, int nhosts, int ntasks)
```

PARAMETERS

bd Bag descriptor. An integer assigned by PvmJobs library.
argv Pointer to array of arguments to the executables
hosts Pointer to array of strings specifying machine names
nhosts Number of hosts in "hosts".
ntasks Number of copies of the executables to start.
numt Number of tasks added to the bag specified by bd.

DISCUSSION

The `bag_addtasks` routine adds `ntasks` copies of the executable to the bag described by `bd`. The name of the executable will be the same as the executable name given to the routine `bag_open`. `bag_addtasks` is useful when the application program needs more slave processes to do a computation.

3.1.9 bag_deltasks

Remove slave processes from the specified bag.

SYNOPSIS

```
int numt = bag_deltasks(const int bd, int *taskid, int ntasks)
```

PARAMETERS

bd Bag descriptor. An integer assigned by PvmJobs library.
ntasks Number of copies of the executables to be removed.
taskid Task ids of processes to be removed from the bag.
numt Number of tasks removed from bag `bd`.

DISCUSSION

The `bag_deltasks` routine removes `ntasks` copies of the executable from the bag described by `bd`. `bag_deltasks` is useful when the application program needs to decrement the number of slave processes or delete dead processes.

3.1.10 bag_jobfirstreq

A slave process is requesting its first job.

SYNOPSIS

```
int cc = bag_jobfirstreq(int *bd, JOB job, PACK_ANS_FN pack_ans,
```

```

                                UNPACK_JOB_FN unpack_job)
    bd      Bag descriptor. An integer assigned by PvmJobs library.
    job     Pointer to a job structure.
    pack_ans Pointer to function which will pack the computed answer.
    unpack_job Pointer to function to unpack received job from master.
    cc      Integer status code returned by the routine.

```

DISCUSSION

The `bag_jobfirstreq` routine is used by a slave process to register with PVM, find out its parent task identifier, receive `bd`, and an initial request for a new job from master process. Application program must define the `unpack_job` function to unpack job, and `pack_ans` function to pack the finished job. `bag_jobfirstreq` employs synchronous communication. It will wait until a new job is received.

3.1.11 bag_jobnextreq

A slave process submits a completed job and requests a new job. **SYNOPSIS**

```

int cc = bag_jobnextreq(const int bd, ANS result, PACK_ANS_FN pack_ans,
                        JOB job, UNPACK_JOB_FN unpack_job)

```

PARAMETERS

```

    bd      Bag descriptor. An integer which identifies the
            bag that the slave process belongs to.
    result  Pointer to an answer structure.
    pack_ans Pointer to function which will pack the computed answer.
    job     Pointer to a job structure.
    unpack_job Pointer to function to unpack received job from master.
    cc      Integer status code returned by the routine.

```

DISCUSSION

The `bag_jobnextreq` routine is used by a slave process to send a completed result to the master process and request a new job from the master process. The argument `bd` is obtained through call to `bag_jobfirstreq`. User's program must define the `unpack_job` function to unpack job and `pack_ans` function to pack the completed result. `bag_jobnextreq` employs synchronous communication. It submits a completed result and waits until a new job arrives.

3.1.12 bag_exit

Tell *PvmJobs* library that application program is leaving PVM.

SYNOPSIS

```

int info = bag_exit( void )

```

PARAMETERS

```

    info  Integer status code returned by the routine.

```

DISCUSSION

The `bag_exit` routine terminates all slave processes in all open bags. Once the routine is sure that all slave processes have exited, it frees up all memory that it has acquired and contacts the `pvm` daemon that this process is leaving PVM. `bag_exit` should be called by the master process before it stops or exits for good.

3.2 PvmJobs Library Usage

A typical scenario for application programming using *PvmJob* library is as follow. A master program will issue a `bag_open` function call to spawn slave processes and notify spawned slave processes of their bag descriptor. A slave process calls `bag_jobfirstreq` to receive the bag descriptor and request its initial job. The master program will use `bag_complete` to receive a completed job or first request of a job from slave process. It then calls `bag_dispatch` to assign a job to the slave process. Once the slave process has completed a job, it will call the `bag_jobnextreq` to submit the result and request a new job. An exemplary diagram of the library usage is shown in Figure 2.

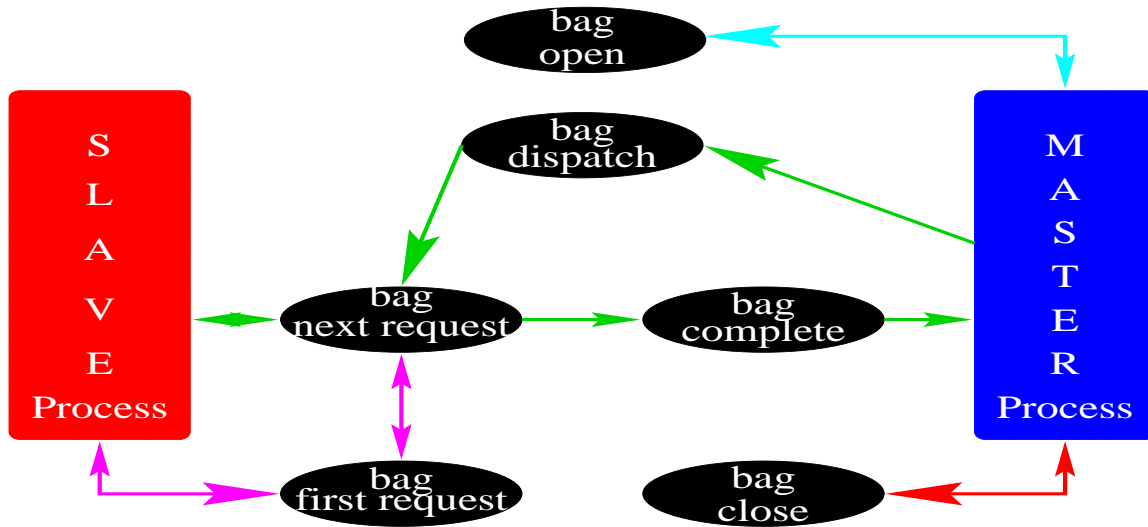


Figure 2: PvmJobs library usage.

4 PvmJobs Example

Next, we present a working example of using *PvmJobs* library to multiply two matrices. In Section 4.1, we present the header file for the matrix program. In Section 4.2 and Section 4.3, we discuss the master and slave program respectively.

4.1 Matrix Multiplication Header File

PvmJobs requires user-defined job and answer structures. We define the job and answer structures for the matrix multiplication program as follows:

```
struct job                                /* job struct that consists of */
{  int jid;                               /* jod id field */
   int *row;                              /* pointer to matrix A row */
```

```

    int rown;                /* index of that row */
    int *col;                /* pointer to matrix B column */
    int coln;                /* index of that column */
    int length;              /* vector length */
};

typedef struct job Job;     /* "Job" type definition */

struct ans                  /* answer struct that consists of */
{
    int ansr;               /* result field */
    int rown;               /* index for a row of matrix A */
    int coln;               /* index for a column of matrix B */
};

typedef struct ans Ans;    /* "Ans" type definition */

```

Also required by *PvmJobs* are user-defined functions to pack and unpack a job and an answer. These are declared as follows:

```

int pack_job (/* j */);    /* job-packing function name */
int unpack_job (/* job */); /* job-unpacking function name */
int pack_ansr (/* a */);   /* answer-packing function name */
int unpack_ansr (/* a */); /* answer-unpacking function name */

```

4.2 Matrix Multiplication Master Program

The master program begins by calling *Pvmjobs* library routine `bag_open()` to create slave processes. Then it calls the library routine `bag_displaytasks()` to display the task identifiers for the spawned slave processes. If there are errors in spawning the slave processes, `bag_displaytasks()` will print the error code in decimal.

```

    if ((bd = bag_open(SLAVENAME, (char **)0, hosts, nhosts, ntasks)) < 0)
    { fprintf(stderr, "\n%s: Please check given host names.\n", argv[0]);
      pvm_exit();
      exit(1);
    }

    printf("Slave tids are:\n");
    bag_displaytasks(bd);

```

Then, we initialize two matrices A and B by randomly generating integers modulo some appropriate modulus.

```

for (r=0 ; r<AROW ; r++)
  for (c=0 ; c<ACOL ; c++)
  { elm =rand()%MODULUS;
    a[r][c] = ( elm > M2 ) ? elm - MODULUS : elm;
  }

```

```

    }
    for (r=0 ; r<BROW ; r++)
        for (c=0 ; c<BCOL ; c++)
            { elm =rand()%MODULUS;
              b[r][c] = ( elm > M2 ) ? elm - MODULUS : elm;
            }
}

```

The master program then initializes the bag of jobs and enter the main loop of dispatching jobs and receiving answers from slave processes. Each job in the bag consists of one row of matrix A and one column of matrix B. A slave process receives a job and returns the dot product of the two vectors.

```

do {
    cc = bag_complete (bd,&slave, &answer, unpack_ansr);

    if (cc > -1)          /* slave has not failed */
    { if (cc == PVMJOBS_YESRESULT)      /* slave is ready */
      {
          jobs_returned++;
          p[answer.rown][answer.coln] = answer.ansr;
      }
      /* dispatch job now */
      if (jobs_in_bag > 0)
      {
          if (bag_dispatch (bd, slave, job_array+n, pack_job) < 0)
          {
              printf("Job dispatch failed..\n");
              bag_exit();
              exit(1);
          }
          jobs_in_bag--;
          n++;
      }
      else
      {
          /* there are no more jobs in job array. */
          if ( bag_dispatch (bd, slave, NULL, pack_job) < 0)
          {
              bag_perror("dispatch NULL job failed..");
              bag_exit();
              exit(1);
          }
      }
    }
    else /* slave failed */
    {

```

```

        bag_perror("bag_open:");
        bag_exit();
        exit(1);
    }
} while (jobs_returned < TOTAL_JOBS ); /* still jobs to farm out */

```

After receiving all jobs from slaves processes, the master program displays the product of the two matrices. It then calls *PvmJobs* library routine `bag_close()` for cleanup and `bag_exit()` to exit PVM.

```

if (bag_close(bd) > -1) /* bag_close was successful */
{
    printf("\nThe jobs facility has been closed normally.\n");
}
else /* bag_close failed due to PVM system error */
{
    bag_perror("The jobs facility has not been closed normally:");
    return;
}
bag_exit();
exit();

```

Now, we present the C codes for user-defined packing and unpacking functions.

```

int pack_job (j)
Job * j;
{
    if (pvm_pkint(&(j->jid), 1, 1) == 0 &&
        pvm_pkint(&(j->length), 1, 1) == 0 &&
        pvm_pkint(j->row, j->length, 1) == 0 &&
        pvm_pkint(j->col, j->length, 1) == 0 &&
        pvm_pkint(&(j->rown), 1, 1) == 0 &&
        pvm_pkint(&(j->coln), 1, 1) == 0
    )
        return 0; /* Packing is successful */
    else
        return -1; /* Packing failed */
}

```

```

int unpack_ansr (a)
Ans * a;
{
    if ( pvm_upkint(&(a->ansr), 1, 1) == 0 &&
        pvm_upkint(&(a->rown), 1, 1) == 0 &&
        pvm_upkint(&(a->coln), 1, 1) == 0
    )
        return 0;
}

```

```

else
    return -1;
}

```

4.3 Matrix Multiplication Slave Program

When the slave program is ready, it calls *PvmJobs* library routine `bag_jobfirstreq` to request its first job and receive its bag descriptor from the master process.

```
cc = bag_jobfirstreq(&bd, &received_job, pack_ansr, unpack_job);
```

The slave process then enters an infinite loop of computing answers, sending results back to the master process, and receiving new jobs. The slave process exits the loop and terminates itself upon receiving a tag `PVMJOBS_NOMOREJOBS`.

```

while ( cc == PVMJOBS_ONEMOREJOBS )
{
    /* Compute answer */
    computed_answer.ansr =
        received_job.row[0] * received_job.col[0];
    for (n=1 ; n < received_job.length ; n++)
        computed_answer.ansr = computed_answer.ansr +
            received_job.row[n] * received_job.col[n];
    computed_answer.rown = received_job.rown;
    computed_answer.coln = received_job.coln;

    /* Send answer back to master and request next job */
    cc = bag_jobnextreq (bd, &computed_answer,pack_ansr,
        &received_job,unpack_job);
}

```

Now, we present the slave packing and unpacking functions.

```

int unpack_job (j)
Job * j;
{
    pvm_upkint(&(j->jid), 1, 1);
    pvm_upkint(&(j->length), 1, 1);
    j->row = (int *) calloc(j->length, sizeof(int));
    pvm_upkint(j->row, j->length, 1);
    j->col = (int *) calloc(j->length, sizeof(int));
    pvm_upkint(j->col, j->length, 1);
    pvm_upkint(&(j->rown), 1, 1);
    pvm_upkint(&(j->coln), 1, 1);
    return(0);
}

```

```

int pack_ansr (a)
Ans * a;
{
  if (pvm_pkint(&(a->ansr), 1, 1) == 0 &&
      pvm_pkint(&(a->rown), 1, 1) == 0 &&
      pvm_pkint(&(a->coln), 1, 1) == 0)
    return 0;
  else
    return -1;
}

```

5 Summary

PvmJobs is designed as a general tool for any PVM application that uses a master/slave message-passing paradigm. It is well documented and easy to use by anyone who knows PVM. Hence, a strong background in PVM is not absolutely necessary. The library has been used in various typical parallel computations and applied in the parallel implementations of the Gröbner Bases Algorithm and the Characteristic Sets Method [1]. The simple yet powerful mechanisms of the library will benefit any users application programs which employ master/slave paradigm. Further enhancements of the *PvmJobs* library are currently undergoing at ICM/Kent. *PvmJobs* will be made available through public FTP.

References

- [1] Iyad A. Ajwa and Paul S. Wang. "Applying Parallel/Distributed Computing to Advanced Algebraic Computations". Submitted for NAECON '97.
- [2] Al Geist *et al.* "PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing". The MIT Press, 1994.
- [3] Paul S. Wang. "Tools to Aid PVM Users". The Fourth U.S. PVM Users' Group Meeting, Santa Fe, New Mexico, USA, February 25-27, 1996.