Parallel Virtual Machines

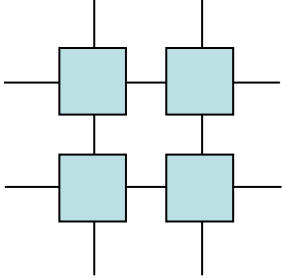
Loosely coupled hetrogeneous virtual multiprocessor

PVM

- Task based
  - Tasks can be created at runtime
  - Tasks can be notified on the death of a parent or child
  - Tasks can be grouped

PVM Architecture

- Demon based communication
- User defined host list
- Hosts can be added and removed during execution
- The virtual machine may be used interactively or in the background



Heterogeneous Computing

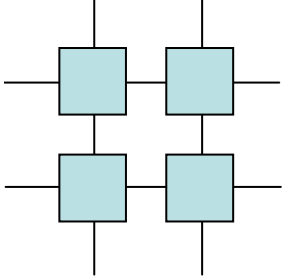
- Runs processes on different architectures
- Handles conversion between little endian and big endian architectures

PVM communication model

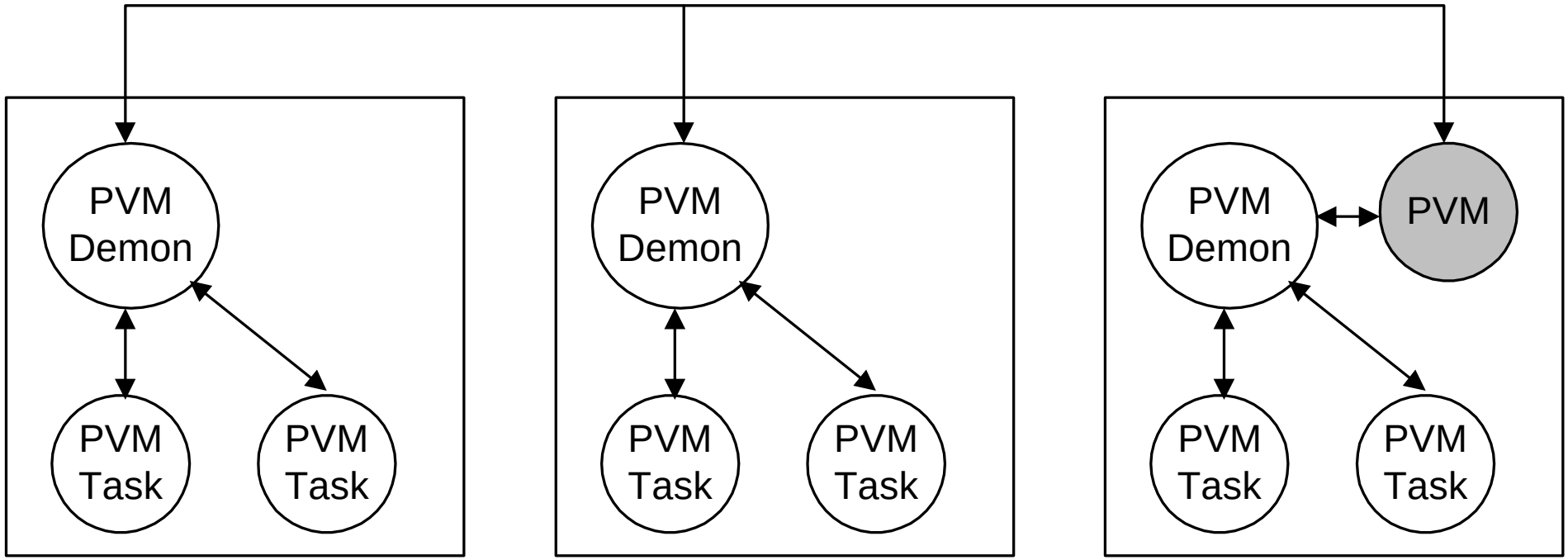
- Explicit message passing
- Has mechanisms for packing into bufferes and unpacking from buffers
- Supports Asynchronous Communication
- Supports one to many communication
  - Broadcast
  - Multicast

The virtual machine codes

- All calls to PVM return an integer, if less than zero this indicates an error
  - pvm_perror();


Cluster Computing

PVM


PVM Demon
PVM Task
PVM Task
PVM Demon
PVM Task
PVM Task
PVM Demon
PVM
PVM Task
PVM Task

Managing the virtual machine

- Add a host to the virtual machine
  - int info = pvm_addhosts(char **hosts, int nhost, int *infos);
- Deleting a host in the virtual machine
  - int info = pvm_delhosts(char **hosts, int nhost, int *infos)
- Shutting down the virtual machine
  - int info = pvm_halt(void);

Cluster Computing

Managing the virtual machine

- Reading the virtual machine configuration
  - int info = pvm config(int *nhost, int *narch, struct pvmhostinfo **hostp)
  - struct pvmhostinfo {
    int hi_tid;
    char *hi_name;
    char *hi_arch;
    int hi_speed;
    } hostp;

Managing the virtual machine

- Check the status of a node
  - int mstat = pvm_mstat(char *host);
    - PvmOk host is OK
    - PvmNoHost host is not in virtual machine
    - PvmHostFail host is unreachable (and thus possibly failed)

Tasks

- PVM tasks can be created and killed during execution
  - id = pvm_mytid();
  - cnt = pvm_spawn(image, argv, flag, node, num, tids);
  - pid = pvm_parrent();
  - pvm_kill(tids[0]);
  - pvm exit();
  - int status = pvm_pstat(tid)

Cluster Computing

Tasks

- int info = pvm_tasks(int where, int *ntask,struct pvmtaskinfo **taskp)

```
struct pvmtaskinfo{
  int ti_tid;
  int ti_ptid;
  int ti_host;
  int ti_flag;
  char *ti_a_out;
  int ti_pid;
} taskp;
```

Managing IO

- In the newest version of PVM output may be redirected to the parent
  - int bufid = pvm_catchout(FILE *ff);

Asynchronous events

- Notifications on special events
  - info = pvm_notify(event, tag, cnt, tids);
  - info = pvm_sendsig(tid, signal);

Groups

- Groups allows for easy fragmentation of the execution in an application
  - num=pvm_joingroup("worker");
  - size = pvm_gsize("worker");
  - info = pvm_lvgroup("worker");
  - int inum = pvm_getinst(char *group, int tid)
  - int tid = pvm_gettid(char *group, int inum)

Buffers

- PVM applications have a default send and a default receive buffer
  - buf=pvm_initsend(Default|Raw|In place);
  - info = pvm_pk(type)(data,10,1);
  - info = pvm_upk(type)(data,10,1);

Managing Buffers

- info = pvm_mkbuffer(Default|Raw|In place);
- oldbuf = pvm_setrbuf(bufid);
- oldbuf = pvm_setsbuf(bufid);
- int info = pvm_freebuf(int bufid)
- int bufid = pvm_getrbuf(void);
- int bufid = pvm_getsbuf(void);

Receiving messages

- Messages may be received blocking or nonblocking
  - bufid = pvm_probe(tid, tag);
  - bufid = pvm_recv(tid, tag);
  - bufid = pvm_trecv(tid, tag, tmout);
  - bufid = pvm_nrecv(tid, tag);
  - info = pvm_precv(tid, tag, array, cnt, type, &atid, &atag, &acnt);

Sending messages

- Messages can also be sent in various ways
  - info = pvm_send(tid, tag);
  - info = pvm_psend(tid, tag, data, cnt, type);

Managing Buffers

- info = pvm_mkbuffer(Default|Raw|In place);
- oldbuf = pvm_setrbuf(bufid);
- oldbuf = pvm_setsbuf(bufid);
- int info = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid);

Global reductions

- Global reductions are useful for a wide array of parallel applications
  - info = pvm_reduce(PvmMax, &data, cnt, type, tag, "workers", roottid);

PVM Reductions

- Global
  - Sum
  - Produkt
  - Min
  - Max

Cluster Computing

PVM Synchronizarions

- Barrier
  - inum=pvm_joingroup("worker");
  - pvm_barrier("worker",5);

Broadcast

- Sends the active buffer to all members of a group
  - info=pvm_bcast(“worker”, 42);
- NOTE: the task that issues a broadcast need not be a member of the group!

Multicasting

- A message can be sent to a number of tasks without the existance of a shared group
  - info = pvm_mcast(list, number, 42);

An example

- Finite differences
- Well know technique for solving differential equations
- The one-dimensional version is trivial if we don’t need information on the evolution in time

Cluster Computing

The model

A system, at time 0, may be represented by a vector

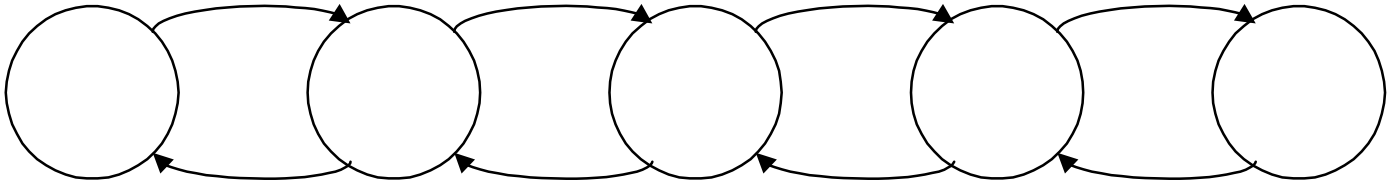
$X^{(0)}=\{x_1, x_2, \ldots , x_n\}$,

At time t+1 each point in the vector is then:

$x^{t+1}_i=(x^t_{i-1}-2 x^t_i+ x^t_{i+1})/h^2$

where h is a discretion constant.

The example



First Solution

If left neighbor exist then

- read data from left
- send data to the left

Update points 0..n-1

If right neighbor exist then

- read data from right
- send data to the right
- update point n

Problems with Solution 1?

- Results in serialization!
- We must eliminate this serialization

Second Solution

If left neighbor exist then

- read data from left
- send data to the left

If right neighbor exist then

- send data to the right
- read data from right
- Update points 0..n

Problems with Solution 2

- Enforced strict synchronous execution
  - Slowest Task dictates progress
- All communication takes place at the same time
  - Stresses the communication network

Solution 3

```
If left neighbor exist then
    send data to the left
If right neighbor exist then
    send data to the right
Update points 1..n-1
If left neighbor exist then
    read data from left
    Update point 0
If right neighbor exist then
    read data from right
    Update points n
```

Problems with solution 3

- Practically none!
- Only potential improvement is to overlap communication and calculation (latency hiding)

Solution 4

If left neighbor exist then

issue_read data from left

issue_send data to the left

If right neighbor exist then

issue_read data from right

issue_send data to the left

Update points 1..n-1

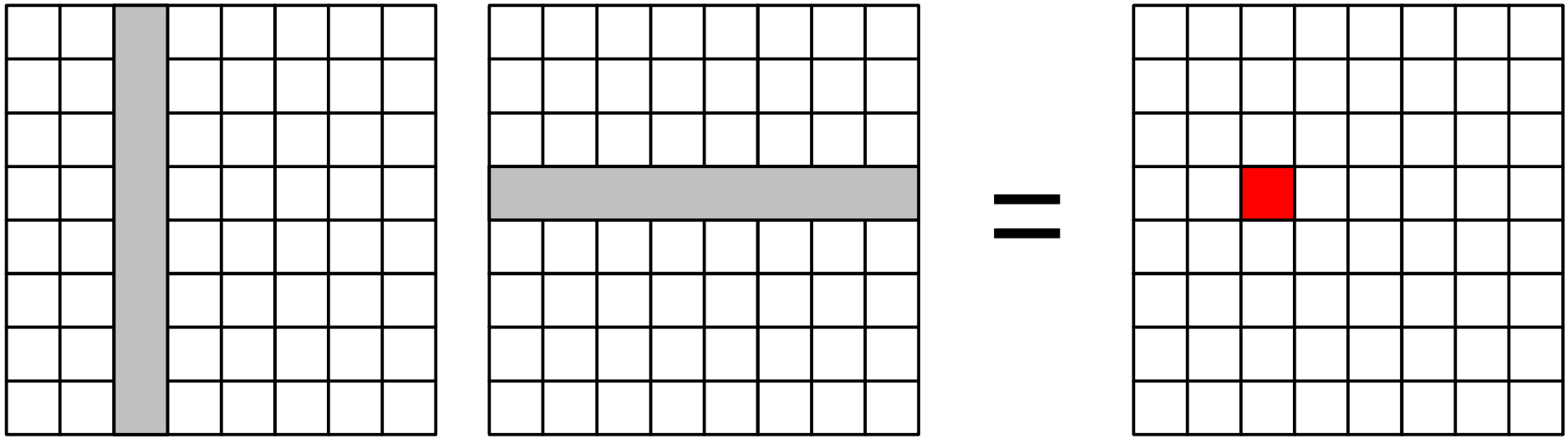
Finish_any_read; Update corresponding point

Finish_any_read; Update corresponding point

Cluster Computing

Matrix Multiplication

Used extremely frequently in scientific applications



Naïve version

```
mxmul(REAL **c, REAL **a, REAL**b, int n)
{
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
      c[i][j]+=a[i][k]*b[k][j]
}
```

The performance of the naïve version may be improved by maintaining B in its transposed form!!

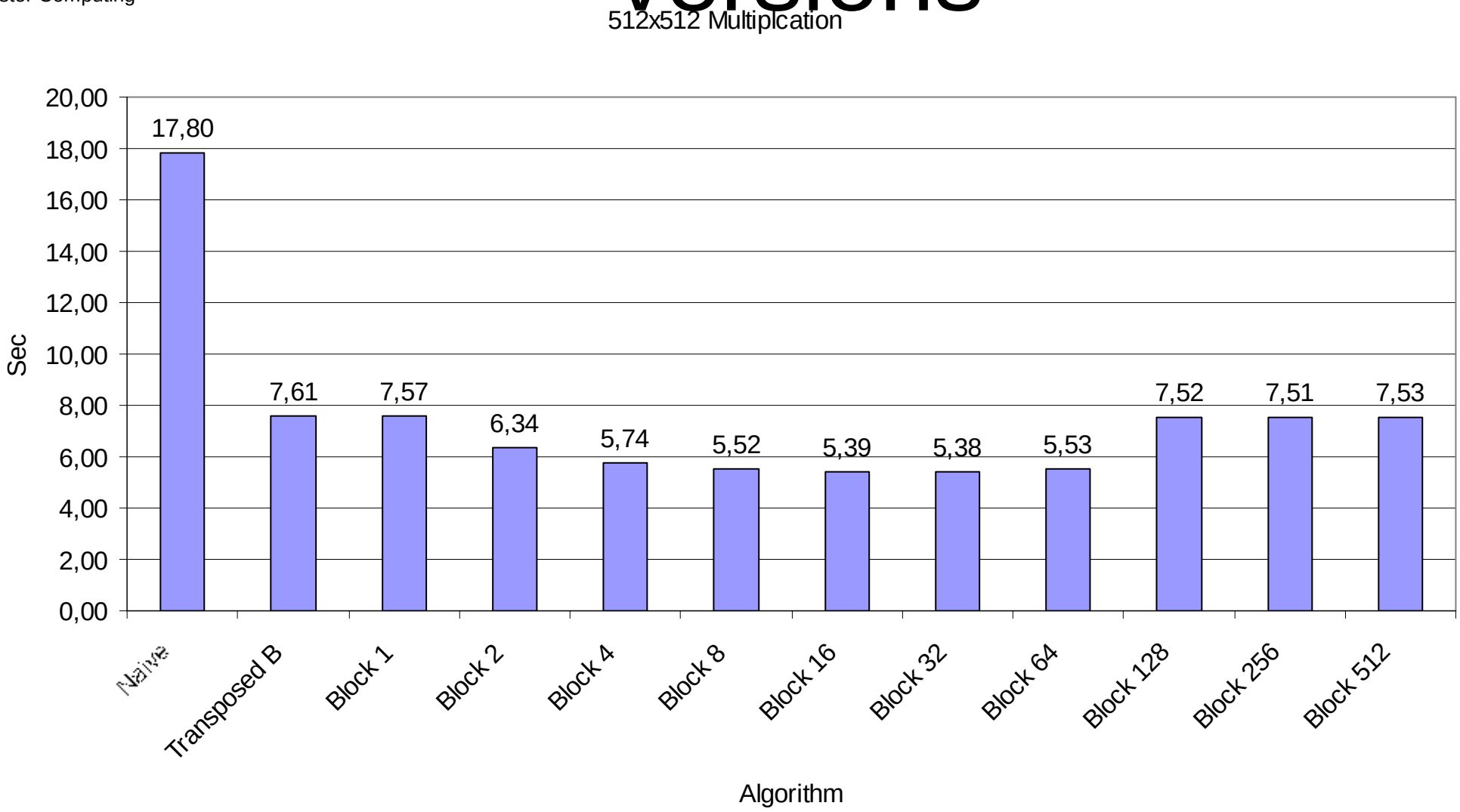
Cluster Computing

Blocked Sequential Version

```
bmul(REAL **c, REAL **a, REAL**b, int is, int js, int bs, int n){
  int i,j,k;

  for(i=is*bs;i<is*bs+bs;i++)
    for(j=js*bs;j<js*bs+bs;j++)
      for(k=0;k<n;k++)
          C(i,j)+=A(i,k)*B(k,j);
}

mxmul(REAL **c, REAL **a, REAL**b, int n){
  int i,j,k;

  for(i=0; i<n; i+=bs)
    for(j=0; j<n; j+=bs)
      bmul(i,i+bs,j,j+bs);
}
```

Cluster Computing

Performace of the Basic versions

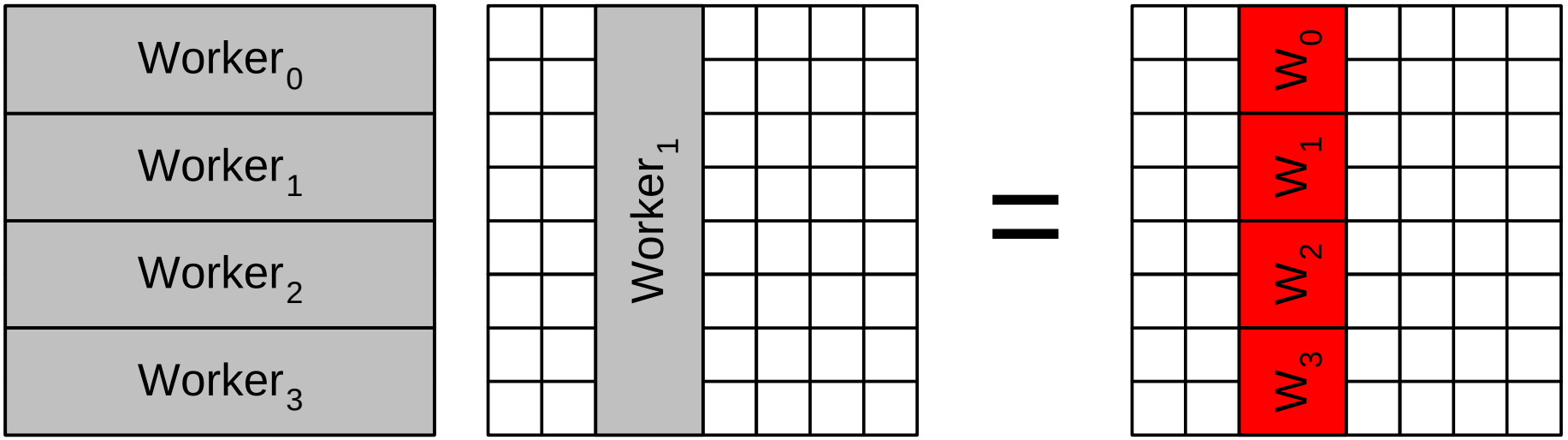


Recursive Version

```
Matrix C mxmul(Matrix A, Matrix B, int s){
  if(s==1)
    C=A*B;
  else {
    s=s/2;
    p0=mxmul(UL(A),UL(B),s);
    p1=mxmul(UR(A),LL(B),s);
    p2=mxmul(UL(A),UR(B),s);
    p3=mxmul(UR(A),LR(B),s);
    p4=mxmul(LL(A),UL(B),s);
    p5=mxmul(LR(A),LL(B),s);
    p6=mxmul(LL(A),UR(B),s);
    p7=mxmul(LR(A),LR(B),s);
    UL(C)=p0+p1; UR(C)=p2+p3;
    LL(C)=p4+p5; LR(C)=p6+p7;
  }
  return C;
}
```

Blocked Parallel Version

- If we have a broadcast media then we can efficiently broadcast blocks to all workers



Blocked Parallel version

- Done in W broadcasts using W workers!

Blocked Version in PVM

- All workers holds one row-block and the corresponding coloum block
- Worker zero first broadcasts its coloum, the one and so forth
- Result is that excatly the size of B is broadcast in *W* blocks

Main

```
main(int argc, char **argv){
  int bs;
  char msg[1024];

  N=atoi(argv[1]); bs=atoi(argv[2]); size=atoi(argv[3]);

  pvm_joingroup("workers");
  rank=pvm_getinst( "workers", pvm_mytid());

  basicBsize=N/size;
  lastBsize=basicBsize+N%size;
  if(rank==size-1)myBsize=lastBsize;
  else myBsize=basicBsize;

  a=(REAL *)malloc(N*lastBsize*sizeof(REAL));
  //same for b,tb and c
  mmul(bs);
  pvm_exit();
}
```

Cluster Computing

Main loop

```
mmul(int bs){
  int w,i,j,k;
  int src, atag, acnt;
  REAL *t=tb;

  for(w=0;w<size;w++){
    pvm_initsend(PVM_COM_MODEL);
    if(rank==w){
      tb=b;
      pvm_pkreal(b, N*(w==size-1 ? lastBsize : basicBsize), 1);
      pvm_bcast("workers", 100+w);
    }
    else {
      pvm_recv(-1,100+w);
      pvm_upkreal(tb,N*(w==size-1 ? lastBsize : basicBsize),1);
}
    for(i=0; i<myBsize; i+=bs)
      for(j=0; j<myBsize; j+=bs)
       bmul(i,i+bs,j,j+bs);
    tb=t;
  }
}
```

How may this version be improved?

- Overlapping communication and calculation

Summary

- PVM is similar to programming with threads - except you need message-passing
- At first parallel programs may be very inefficient
- More efficient programs are more complex