Advanced parallel programming

Dottorato di Ricerca
Dip. Informatica di Pisa

M. Danelutto
Giugno-Luglio 2007

Program

- Introduction
- Classical programming models
- Structured programming models
- Skeleton parallel programming environments
- Open problems
- Project

Summarizing ...

Modelli di programmazione ("*di produzione*")

- Message passing puro
  - PVM 1989 (Univ. Tennessy, Emory and Oak Ridge Nat. Lab.)
  - MPI (Draft 1.0 1994)
    - SPMD
- RPC
  - Sun RPC, Java RMI, CORBA,
- Shared memory
  - HPF (1993)
  - OpenMP (1997)

M. Danelutto - Tecniche di programmazione avanzata - Corso di dottorato - Pisa - Giu-Lug-07

PVM

- Parallel Virtual Machine (1989)
  - configurable host pool: heterogeneous hsts, added/removed dynamically
  - virtual host (possibly) exploiting exploit peculiar features of PEs (mapping)
  - base element: process
    - mapping decided independently (processes to processors)
- heterogeneous processing elements support
  - data forma
  - network independency

Sample code

```
main()
{
    int cc, tid, msgtag;
    char buf[100];

    printf("i'm t%x\n", pvm_mytid());

    cc = pvm_spawn("hello_other", (char**)0, 0, "", 1,
&tid);

    if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
    } else
      printf("can't start hello_other\n");

    pvm_exit();
}
```

Sample code (2)

```
#include "pvm3.h"

main()
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
     msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);

    pvm_exit();
}
```

Virtual machine concept

- compile programs with its own script/libraries
- pvm console
- host handling
  - add hostname, delete hostname
- management
  - conf, halt
- non interactive command launching
  - spawn

Communication primitives

- int bufid = pvm_initsend(int encoding)
  - declares a buffer (XDR, without encoding, or pointer only)
- int info = pvm_pkint(int   *np, int nitem, int stride)
  int info = pvm_pkdouble(double *dp, int nitem, int stride)
  int info = pvm_pkstr(char   *cp)
  int info = pvm_packf(const char *fmt, ...)
  - prepare data in buffer
- int info = pvm_send(int tid, int msgtag)
  - send buffer
- int info = pvm_mcast(int *tids, int ntask, int msgtag)
  - broadcast / multicast on tids

Communication primitives (2)

- int bufid = pvm_recv(int tid, int msgtag)
  - buffer receive
- int info = pvm_upkdouble(double *dp, int nitem, int stride)
  int info = pvm_upkint(int *np, int nitem, int stride)
  int info = pvm_upkstr(char *cp)
  - de-packetization
- may have message tags
  - separately handled

Process handling

- int tid = pvm_mytid(void)
  - process rank in virtual machine
- int info = pvm_exit(void)
  - process termination

- int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)
  - create a number of processes
- int info = pvm_kill(int tid)
  - terminate a process

Process handling (2)

- int info = pvm_catchout(FILE *ff)
  - catch output of “next spawned” processes
- int info = pvm_addhosts(char **hosts, int nhost, int *infos)
  int info = pvm_delhosts(char **hosts, int nhost, int *infos)
  - add/remove host
- int info = pvm_sendsig(int tid, int signum)
  - send signal (POSIX)

Sample code: matrix multiplication

```
int
main(int argc, char* argv[])
{

    /* number of tasks to spawn, use 3 as the default */
    int ntask = 2;
    /* return code from pvm calls */
    int info;
    /* my task and group id */
    int mytid, mygid;
    /* children task id array */
    int child[MAXNTIDS-1];
    int i, m, blksize;
    /* array of the tids in my row */
    int myrow[MAXROW];
    float *a, *b, *c, *atmp;
    int row, col, up, down;

    /* find out my task id number */
    mytid = pvm_mytid();
    pvm_advise(PvmRouteDirect);

    /* check for error */
    if (mytid < 0) {
        /* print out the error */
        pvm_perror(argv[0]);
        /* exit the program */
        return -1;
        }

    /* if my group id is 0 then I must spawn the other tasks */
    if (mygid == 0) {
        /* find out how many tasks to spawn */
        if (argc == 3) {
            m = atoi(argv[1]);
            blksize = atoi(argv[2]);
            }
        if (argc < 3) {
            fprintf(stderr, "usage: mmult m blk\n");
            pvm_lvgroup("mmult"); pvm_exit(); return -1;
            }

        /* make sure ntask is legal */
        ntask = m*m;
        if ((ntask < 1) || (ntask >= MAXNTIDS)) {
            fprintf(stderr, "ntask = %d not valid.\n", ntask);
            pvm_lvgroup("mmult"); pvm_exit(); return -1;
            }
        /* no need to spawn if there is only one task */
        if (ntask == 1) goto barrier;

        /* spawn the child tasks */
        info = pvm_spawn("mmult", (char**)0, PvmTaskDefault, (char*)0,
            ntask-1, child);

        /* make sure spawn succeeded */
        if (info != ntask-1) {
            pvm_lvgroup("mmult"); pvm_exit(); return -1;
            }
        /* send the matrix dimension */
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&m, 1, 1);
        pvm_pkint(&blksize, 1, 1);
        pvm_mcast(child, ntask-1, DIMTAG);
        }
```

Sample code (2)

```
    else {
        /* recv the matrix dimension */
        pvm_recv(pvm_gettid("mmult", 0), DIMTAG);
        pvm_upkint(&m, 1, 1);
        pvm_upkint(&blksize, 1, 1);
        ntask = m*m;
        }
    /* make sure all tasks have joined the group */
barrier:
    info = pvm_barrier("mmult",ntask);
    if (info < 0) pvm_perror(argv[0]);

    /* find the tids in my row */
    for (i = 0; i < m; i++)
        myrow[i] = pvm_gettid("mmult", (mygid/m)*m + i);

/* allocate the memory for the local blocks */
a = (float*)malloc(sizeof(float)*blksize*blksize);
b = (float*)malloc(sizeof(float)*blksize*blksize);
c = (float*)malloc(sizeof(float)*blksize*blksize);
atmp = (float*)malloc(sizeof(float)*blksize*blksize);
/* check for valid pointers */
if (!(a && b && c && atmp)) {
    fprintf(stderr, "%s: out of memory!\n", argv[0]);
    free(a); free(b); free(c); free(atmp);
    pvm_lvgroup("mmult"); pvm_exit(); return -1;
    }
```

```
/* find my block's row and column */
row = mygid/m; col = mygid % m;
/* calculate the neighbor's above and below */
up = pvm_gettid("mmult", ((row)?(row-1):(m-1))*m+col);
down = pvm_gettid("mmult", ((row == (m-1))?col:(row+1)*m+col));

/* initialize the blocks */
InitBlock(a, b, c, blksize, row, col);
/* do the matrix multiply */
for (i = 0; i < m; i++) {
    /* mcast the block of matrix A */
    if (col == (row + i)%m) {
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(a, blksize*blksize, 1);
        pvm_mcast(myrow, m, (i+1)*ATAG);
        BlockMult(c,a,b,blksize);
        }
    else {
        pvm_recv(pvm_gettid("mmult", row*m + (row +i)%m), (i+1)*ATAG);
        pvm_upkfloat(atmp, blksize*blksize, 1);
        BlockMult(c,atmp,b,blksize);
        }
    /* rotate the columns of B */
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(b, blksize*blksize, 1);
    pvm_send(up, (i+1)*BTAG);
    pvm_recv(down, (i+1)*BTAG);
    pvm_upkfloat(b, blksize*blksize, 1);
    }
/* check it */
for (i = 0 ; i < blksize*blksize; i++)
    if (a[i] != c[i])
        printf("Error a[%d] (%g) != c[%d] (%g) \n", i, a[i],i,c[i]);

printf("Done.\n");
free(a); free(b); free(c); free(atmp);
pvm_lvgroup("mmult");
pvm_exit();
return 0;
}
```

Debugging

- Unix processes, therefore gdb attach/debug
- but :

Debugging and Tracing

First, the bad news. Adding `printf()` calls to your code is still a state-of-the-art methodology.

Code tangling

- functional code with inner code for
  - process setup/management
  - communications
- functional debugging is impossible
  - common feature for all the classical message passing environments

MPI

- Standard Messagge Passing Interface
  - First version in 1994
  - “expert” committee
    - University, industry, research institutions
  - more than one hundred calls
    - most used = O(10)
  - language independent
  - defines standard, not the implementation
  - Second version (1.2) adds one side comms, I/O e dynamic processes

Top features

- provides
  - virtual process topology (process *is not* a processor)
  - comms and synchro
    - point to point and collective (broadcast, scatter, gather)
      - sycnhronous and asynchronous
      - collective operations (reduce, barrier)
- different implementations
  - open source (MPICH, LAM) or vendor/proprietarie
  - available in Unix/Linux/BSD, Windows, ...

Top Features (2)

- communicator
  - group of processes + identifiers
  - used for collectives
  - MPI_COMM_WORLD (all processes)
    - can be split
  - point to point comms inter and intra communicator

- SPMD model
  - Single Program Multiple Data

Main MPI

```
#include "mpi.h"

int main(int argc, char * argv[]) {


  int world_size, my_id;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

  if(my_id == 0) {
    /* do something */
  } else {
    /* do something else */
  }

  MPI_Finalize();
}
```

Sample code: master-slave

- id == 0
  - master
  - round robin point to point on slaves to deliver tasks
  - idem to collect results
- id != 0
  - get task
  - compute
  - send answer

Point to point comms

- Blocking send + receive
  - int MPI_Send(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)
- Non blocking
  - int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
  - and then a number of variants (w.r.t. buffer usage, ...)

Envelope

- Message:
  - data (MPI_some_type o MPI_BYTE)
  - tag
    - integer
      - matching with tag or MPI_ANY_TAG in the receive

- MPI data types (wrappings)
  - guarantee interoperability between different machines (automatic marshalling/unmarshalling)

Collective (1)

- Broadcast
  - int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Gather
  - int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- Scatter
  - int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- vector versions (in addition)

Collective (2)

- Reduce
  - int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - MPI_Op
    - MPI_MAX, MPI_MIN, MPI_ADD, ... MPI_LAND, MPI_BAND, ...
    - int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
- Scan
  - int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Groups and topologies

- can define new groups
  - as a partition of initial group
  - with inter and intra communications

- can define virtual topologies
  - cartesian product, generic graph
  - with opportune process index schemas

Full sample code

```
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"


main(int argc, char** argv) {
   int        my_rank;   /* My process rank           */
   int        p;         /* The number of processes   */
   double     a = 0.0;   /* Left endpoint             */
   double     b = 1.0;   /* Right endpoint            */
   int        n = 1024;  /* Number of trapezoids      */
   double     h;         /* Trapezoid base length     */
   double     local_a;   /* Left endpoint my process  */
   double     local_b;   /* Right endpoint my process */
   int        local_n;   /* Number of trapezoids for  */
                         /* my calculation            */
   double     integral;  /* Integral over my interval */
   double     total;     /* Total integral            */
   int        source;    /* Process sending integral  */
   int        dest = 0;  /* All messages go to 0      */
   int        tag = 0;
   MPI_Status  status;

/* Fall 97:  Wall Time Declarations:  */
      double       start, finish, totalwalltime, traperror;

   double Trap(double local_a, double local_b, int local_n,
            double h);    /* Calculate local integral  */
```

```
/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
   start = MPI_Wtime();

h = (b-a)/n;    /* h is the same for all processes */
local_n = n/p;  /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
if (my_rank == 0) {
   total = integral;
   for (source = 1; source < p; source++) {
      MPI_Recv(&integral, 1, MPI_DOUBLE, source, tag,
         MPI_COMM_WORLD, &status);
      total = total + integral;
   }
} else {
   MPI_Send(&integral, 1, MPI_DOUBLE, dest,
      tag, MPI_COMM_WORLD);
}
   finish = MPI_Wtime();

/* Print the result */
if (my_rank == 0) {
   printf("With n = %4d trapezoids, our estimate\n", n);
   traperror = total - 1.0e0;
   printf("of the integral on (%6.2f,%6.2f) = %11.5f; Error = %12.4e\n",
      a, b, total, traperror);
   totalwalltime=finish - start;
   printf("Integration Wall Time =%9.6f Seconds on %4d Processors\n",
        totalwalltime, p);
}

   MPI_Finalize();
} /*  main  */
```

```
double Trap(
      double  local_a   /* in */,
      double  local_b   /* in */,
      int    local_n   /* in */,
      double  h         /* in */) {
   double integral;   /* Store result in integral  */
   double x;
   int i;

   double f(double x); /* function we're integrating */

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap  */
```

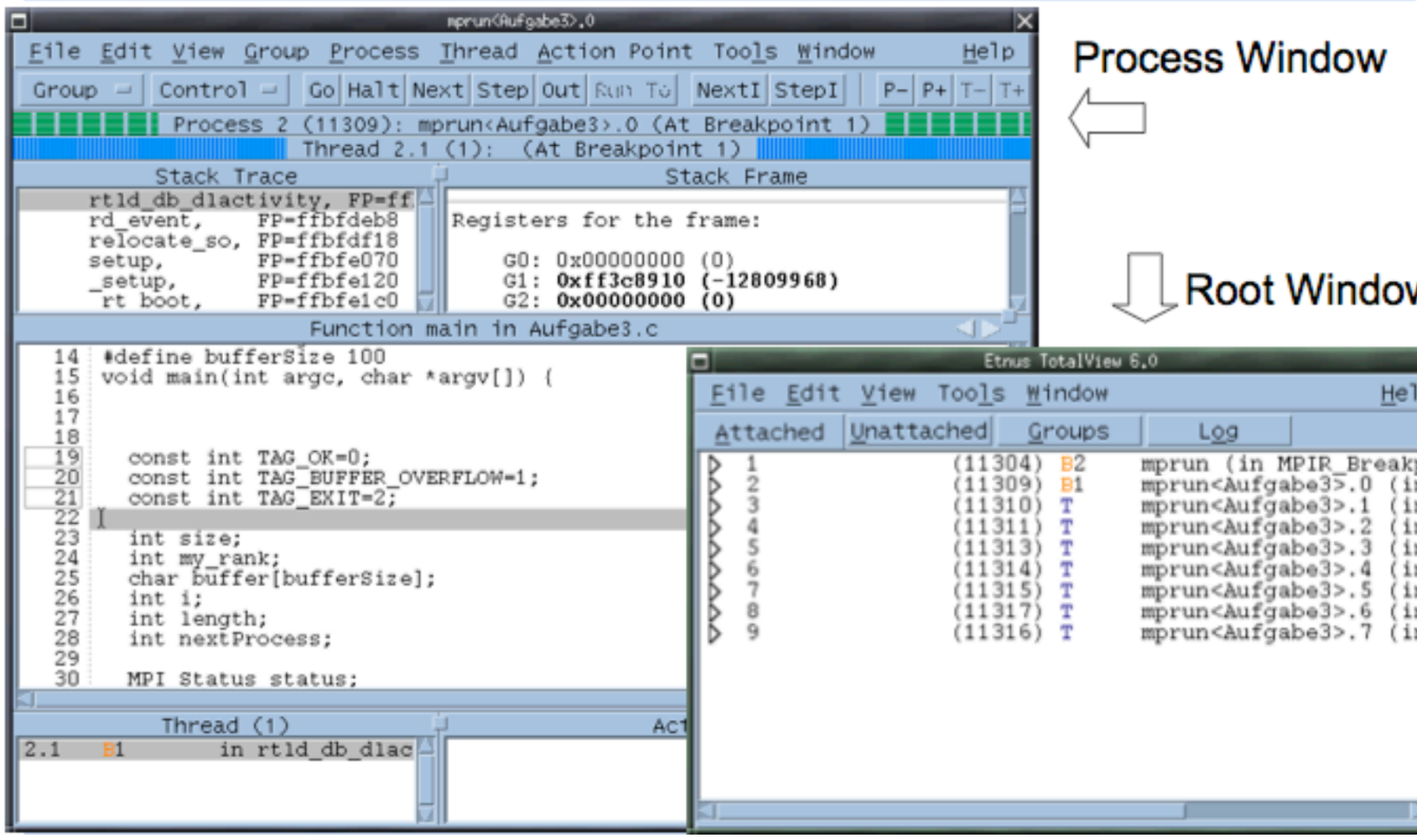
```
double f(double x) {
   double return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = 5*x*x*x*x;
   return return_val;
} /* f */
```

Problems

- full programmer responsibility
  - define processes (number, semantics)
  - comm/synchro structure
  - deploying
    - **mpirun -np N** on architecture with M < N processors: round robin
      - M = N/2 may be better than M = N
- Code tangling
  - functional and non functional code at the same level
- Debugging
- Fine Tuning

Debugging

- “distributed” and graphic debugger available (e.g. TotalView)
- more often: gdb on archie nodes (attach + debug)



Pro

- simple compile and run commands
  - mpicc
  - mpirun [-np nn] [-machinefile filename]

- simple and efficient data parallel computations

- arbritrary complex data parallel patterns

MPI 1.2

- Comunicazioni one-way
  - remote accesses
- dynamic process handling
  - spawn of new processes
- I/O
  - parallel file system access

Process spawn

- int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
  - tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is collective over comm, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the as the ordering of the group of the comm in the parents and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

Spawn sample

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   if (world_size != 1)    error("Top heavy with management");

   MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                &universe_sizep, &flag);
   if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
        scanf("%d", &universe_size);
   } else universe_size = *universe_sizep;
   if (universe_size == 1) error("No room to start workers");

    choose_worker_program(worker_program);
   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
             MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
             MPI_ERRCODES_IGNORE);
   /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0,..
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
    * "everyone".
    */

   MPI_Finalize();
   return 0;
}
```

Spawn sample (2)

```
/* worker */

#include "mpi.h"
int main(int argc, char *argv[])
{
   int size;
   MPI_Comm parent;
   MPI_Init(&argc, &argv);
   MPI_Comm_get_parent(&parent);
   if (parent == MPI_COMM_NULL) error("No parent!");
   MPI_Comm_remote_size(parent, &size);
   if (size != 1) error("Something's wrong with the parent");

   /*
    * Parallel code here.
    * The manager is represented as the process with rank 0 in (the remote
    * group of) MPI_COMM_PARENT.  If the workers need to communicate among
    * themselves, they can use MPI_COMM_WORLD.
    */

   MPI_Finalize();
   return 0;
}
```

One side comms

- Put
  - int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
  - The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, and comm is a communicator for the group of win.
- Get
  - int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

One way synchronization

- int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
  - solo rank lavora su win RMA
- int MPI_Win_unlock(int rank, MPI_Win win)
  - completa tutte le operazioni in win

- void MPI::Win::Fence(int assert) const
  - The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

I/O

- file system access, POSIX style
- local buffering (performance)
- collective open primitive
    - each process has its own view
- access patterns
    - one element read two skipped, ...
- initial displacement
- consequent access

HPF

- FORTRAN dialect derived from F90
  - defines a grid of processes/processors
  - distribute data onto grids
  - parallel computations
    - FORALL (INDEPENDENT)
    - PURE function/procedures
    - EXTRINSIC (external code)
- SIMD model
  - without explicit communication primitives

Directives

- All HPF are directives
  - FORTRAN comments
  - can be ignored by compiler (all, part of, this is important!!!!)
- general form
  - !HPF$ <directive>
- Include:
  - processor grid declaration
  - data distribution
  - mark “parallel” code (statements)

Processor grids

- !HPF$ PROCESSORS, DIMENSION(10) :: A
  - string of processors, named A
- !HPF$ PROCESSORS, DIMENSION(10,10) :: B
  - three dimensional processor array
- logical grid
  - machine dependent adaptation (current number of resources vs. number of processors declared in the program)
    - implementation dependent

Data distribution

- ```
        REAL DIMENSION(100,100) :: X,Y,Z
  !HPF$ PROCESSOR, DIMENSION(10,10) :: A
  !HPF$ DISTRIBUTE (BLOCK) ONTO A :: X
  !HPF$ DISTRIBUTE (BLOCK,CYCLIC) ONTO A :: Y
  !HPF$ DISTRIBUTE (BLOCK,*) ONTO A :: Z
  ```
  - distributions
    - **BLOCK** diveded in (contiguous blocks)
    - **CYCLIC** round robin
    - * all the dimension to one processor
- can be combined on dimensions
- stride/partition dimensions determined at run time depending on #PE
- !HPF$ ALIGN aligns different arrays to the topology
  ```

Parallel computation: FORALL

- statement FORALL (derived from F95)

  FORALL (i=1:N, j=1:M, X(i,j).NE.0) X(i,j) = -3 * X(i,j)

  - all iterations performed in parallel
  - each processor has its own portion!
  - may call PURE functions in the body
    - PURE functions: no side effect, I/O, calls to ESTRINSIC, ...
  - may use idexes in RHS

Parallel computation: INDEPENDENT

- !HFP$ INDEPENDENT

  - can be applied to DO e FORALL, sets up independent iterations

  - !HPF$ INDEPENDENT
    DO I=1,N
      X(I) = I * 200
    END DO

  - iterations computed in parallel

    - *owner computes rule*

- FORALL with !HPF$ INDEPENDENT may evaluate RHS e LHS in parallel!

Sample code: gaussian elimination

```
subroutine gauss(n,A,x)
real A(n,n+1), x(n), fac(n), row(n+1), maxval
integer index(n), itemp(1)
integer i,j,k,n, max_index
index=0                                          ! Initialise array
do i = 1,n                                       ! Repeat for each column
! find pivot
    itemp = MAXLOC(ABS(A(:,i)), MASK=index==0)
    max_index = itemp                            !Extract pivot index
    index(max_index) = i                         ! Update array
    fac = A(:,i)/A(max_index,i)                  !Calc scale factors
    row = A
(max_index,:)
    ! Extract pivot row
! row update
    FORALL(j=1:n, k=i:n+1, index(j)==0) &
        A(j,k) = A(j,k) - fac(j) * row(k)
enddo
FORALL(j=1:n) A(index(j),:)=A
(j,:)
                !row exchange
do j = n,1,-1                                    ! back substitution
    x(j) = A(j,n+1)/A(j,j)
    A(1:j-1,n+1) = A(1:j-1,n+1) - A(1:j-1,j)*x(j)
enddo
end
```

Pros & cons

- agile for SPMD data parallel
  - definitely close to FORTRAN philosophy !!!
- control parallel: nothing to do !
  - even a simple master/slave is an hard task to program
- out of the scope of data parallel :
  - MPI or OpenMP bindings ...
    - again: all in charge to programmers ...

The “HPF plot”

- slow startup
  - a few compilers, very expensive
  - one open source compiler (Adaptor, actually HPF -> MPI)
  - good performance iff full matching program/architecture

- then
  - MPI standard de facto even with FORTRAN
  - but also new and better compilers
    - Europe contribution: Vienna and Paris (Ecole de mines)

OpenMP

- API for writing multithreaded applications
  - set of compiler directives
  - with bindings in C, C++, FORTRAN
  - supported/supplied by major vendors: Intel, HP, SGI, SUN, Compaq
- fork-join parallelism
  - forks a number of threads then join ...
- usually used to parallelize loops (with threads)
- shared memory : synchronization required to handle non standard cases

Directives

- C, C++
  - #pragma omp <*directive*>
- FORTRAN
  - C$OMP <*directive*>
- *comments* therefore
  - OMP programs can be executed as normal programs
    - easy functional debugging

Parallel regions

- sample C code

```
omp_set_num_threads(10);
 #pragma omp parallel
{
   int id = omp_thread_num();
   ... (id, shared_data_X);
}
do_something_else();
```

- spawns 10 threads
  - access to **shared_data_x** are shared (be careful!)
  - access to **id** are local
  - barrier before continuation (before do_something_else();)

Work sharing

- parallel computation of loop iterations

```
#pragma omp parallel
#pragma omp for
for(i=0; i<N; i++) {
    do_something(i, ...);
}
```

- barrier at the end of the for

- possibility to use **schedule** options to modify scheduling of iterations to threads

    - **#pragma omp for schedule(static,chunck)**
    - **static,chunck** chunck items per thread, static
    - **dynamic,chunck** chunkc items per thread, grabbed
    - ...

Data sharing model

- SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE
  - shared,
  - private to threads
  - private, get initial value
  - private, send back last value
- **#pragma omp parallel for private(B)**
  for(...) {
  ... use B as private (to thread) variable ...
  }

Reduction

- **#include <omp.h>**
  #define NUM_THREADS 2
  void main ()
  {
  inti;
  double ZZ, func(), res=0.0;
  omp_set_num_threads(NUM_THREADS)
  #pragma omp parallel for reduction(+:res) private(ZZ)
  for (i=0; i< 1000; i++){
  ZZ = func(I);
  res= res+ ZZ;
  }
  }

- sums up values to the final res value

Synchronization

- #pragma omp critical
  - defines section entered by only one thread
- #pragma omp atomic
  - defines critical section w.r.t. memory accesses
- #pragma omp barrier
  - all threads synchronize
- #pragma omp ordered
  - enforces sequential ordering on a block (w.r.t. threads)
- #pragma omp master
  - defines block only executed by master
- #pragma omp flush(varname)
  - synchronizes shared mem

Pros (from wikipedia)

- Simple: need not deal with message passing as MPI does
- Data layout and decomposition is handled automatically by directives.
- Incremental parallelism: can work on one portion of the program at one time, no dramatic change to code is needed.
- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.
- Both coarse-grained and fine-grained parallelism are possible

Cons (from wikipedia)

- Currently only runs efficiently in shared-memory multiprocessor platforms
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- Reliable error handling is missing.
- Lack fine-grain mechanisms to control thread-processor mapping.
- Synchronization between a subset of threads is not allowed.

Tools

- Compiler tools (merged in GCC 4.1)
  - compiling source code + OMP to some multithreading code
  - with different level of optimizations
    - including loop to OMP processing

Conclusion

- OpenMP is:
  - A great way to write fast executing code.
  - Your gateway to special, painful errors.
- You can save yourself grief if you consider the possible danger zones as you write your OpenMP programs.
- Tools and/or a discipline of writing portable sequentially equivalent programs can help.