Scientific Computing on Parallel Machines

Cálculo científico en ordenadores paralelos

Markus Uhlmann

CIEMAT

www.ciemat.es/sweb/comfos/personal/uhlmann

UCM – Abril 2008

Schedule

Part I	Introduction to parallel programming	lecture 1
Part II	Introduction to MPI	
	General introduction	
	& MPI program structure	lecture 2
	Point-to-point communication	lecture 3,4
	Collective communication	lecture 5
	2D Poisson example	lecture 6,7
	Non-contiguous data & Mixed datatypes	lecture 7
	Virtual topologies & Communication subsets	lecture 8
	Use of linear algebra libraries	lecture 9
	Extensions	lecture 10

Part II Introduction to MPI

Non-contiguous data & derived datatypes

Non-contiguous data & derived datatypes

Previously covered contiguous data, single type messages

Need for communication of non-contiguous data

- Fortran arrays stride along first index
  `a(1,1),a(2,1),a(3,1),a(1,2),a(2,1),...`
- C arrays stride along last index
  `a[0][0],a[0][1],a[0][2],a[1][0],a[1][1],...`

Often useful to communicate multiple types, one message

- combination of integer/real variables
- general `structs`

Non-contiguous data & derived datatypes

Possible solutions

- sending multiple messages
- buffering
- packing the data
- defining derived data types

Sending multiple messages

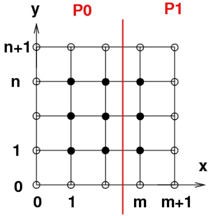
Finite-difference example

- global `A(0:m+1,0:n+1)`
- local `A(ib-1:ie+1,0:n+1)`
- exchange columns of data `A(ib,:)`
- non-contiguous! (in Fortran)
- solution:

```
do j=0,n+1
  call MPI_SEND(A(ib,j),1,...)
enddo
```

- many messages → expensive (latency)

1D in x



Alternative: simple buffering

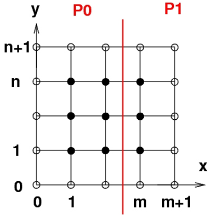
Copy data into a contiguous buffer

- solution:

```
do j=0,n+1
  buff(j)=A(ib,j)
enddo
call MPI_SEND(buff,n+2,...)
```

- eliminates multiple messages
- cost of copying

1D in x



MPI Packing

Pack data into a contiguous buffer

- solution:

```
position=0
do j=0,n+1
  call MPI_PACK(A(ib,j),1,MPI_REAL,
    buff, buffsize,position, MPI_COMM_WORLD,ierr)
enddo
ncount=position
call MPI_SEND(buff,ncount,MPI_PACKED,...)
```

- `buffsize`, `position` are in bytes!
- `position` is incremented from 0 (in/output argument)

MPI Packing – continued

Can pack different objects into same buffer

- ```
  call MPI_PACK(A,1,MPI_REAL,buff,...)
  call MPI_PACK(n,1,MPI_INTEGER,buff,...)
  ```

Determining the max. required size in bytes

- ```
  MPI_PACK_SIZE(ndat,datatype,MPI_Comm,
  buffsize,ierr)
  ```

Unpacking the buffer (receiving processor)

- ```
  MPI_UNPACK(buffer,buffsize,position,
  outbuff,outcount,datatype,MPI_Comm,ierr)
  ```
  ```

MPI Packing vs. simple buffering

MPI Packing

- allows for different datatypes, single buffer
- size of "encoded" data is in general larger than native size

Simple buffering

- often sufficient solution for single types

MPI derived datatypes

Datatype constructors

▶ Contiguous	subsequent elements
▶ Vector	blocks with regular strides
▶ Hvector	as 'Vector', strides in bytes
▶ Indexed	arbitrarily spaced blocks
▶ Hindexed	as 'Indexed', displacement in bytes
▶ Struct	blocks can be of different type

Vector

Create a vector (strided) datatype

- `call MPI_TYPE_VECTOR(ncount,nblocklen,nstride, MPI_REAL,vec_type,ierr)`
- `ncount` – number of blocks
- `nblocklen` – number of elements in each block
- `nstride` – stride between start of each block

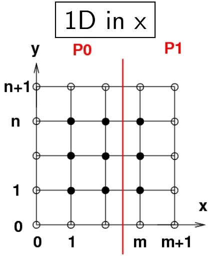
Commit the new datatype

- `call MPI_TYPE_COMMIT(vec_type,ierr)`
- declares new datatype for use in communication routines
- committing is independent of content

Vector datatype solution to example

Communication of columns of `A(ib-1:ie+1,0:n+1)`

```
call MPI_TYPE_VECTOR(n+2,1,ie-ib+3,
  MPI_REAL,vec_type,ierr)
call MPI_TYPE_COMMIT(vec_type,ierr)
call MPI_SEND(A(ib,0),1,vec_type,...)
```



- datatypes are re-usable
- deallocate user-defined datatypes with `call MPI_TYPE_FREE(vec_type,ierr)`

MPI struct datatype

Create a struct datatype

- `call MPI_TYPE_STRUCT(ncount,nblocklen,disp, types,struc_type,ierr)`
- `ncount` – number of blocks
- `nblocklen` – integer array with # of elements in each block
- `disp` – array of byte displacements at start of blocks (measured from beginning of input buffer in send/recv)
- `types` – array of data types

⇝ provides full flexibility

Struct datatype solution to example

Communication of columns of `A(ib-1:ie+1,0:n+1)`

- ```
  integer lena(0:n+1),disp(0:n+1),typea(0:n+1)
  do j=0,n+1
    lena(j)=1
    call MPI_ADDRESS(A(ib,j), disp(j)iaddr ,ierr)
    disp(j)=iaddr-disp(0)
    typea(j)=MPI_REAL
  enddo
  call MPI_TYPE_STRUCT(n+2,lena,disp,
       typea,struct_type,ierr)
  call MPI_TYPE_COMMIT(struct_type,ierr)
  call MPI_SEND(MPI_BOTTOM A(ib,0),1,struct_type,...)
  ```
- using absolute addresses relative to MPI_BOTTOM using relative addresses
  ```

Packing vs. derived datatypes

Advantages of packing

- allows for incremental unpacking:
  e.g. beginning of message determines extent and type of following data

$\longrightarrow$ useful for complex, dynamic data layouts

Advantages of derived datatypes

- avoids defining intermediate buffers (memory)
- often faster than packing (typedef only called once)

$\longrightarrow$ useful for static or regular data layouts

Exercise: parallel search problem

Problem statement

- search entries in a large table
- find indices and average: `(index+target)/2`
- transmit 1 integer & 1 real value per match

Strategy

- modify previous code by using `MPI_PACK` or `MPI_TYPE_STRUCT`
  (previous source) (input data)

Code example: parallel search

Correct result:

```
P 1 62 36.
P 3 271 140.5
P 3 291 150.5
P 3 296 153.
P 2 183 96.5
```

(source with packing) (struct)