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

Case Study: Jacobi Iteration of 2D Poisson Problem

$$\begin{aligned} (\partial_{xx} + \partial_{yy})\, u &= 0 \\ u(\mathbf{x} \in \Gamma) &= u_\Gamma \end{aligned}$$

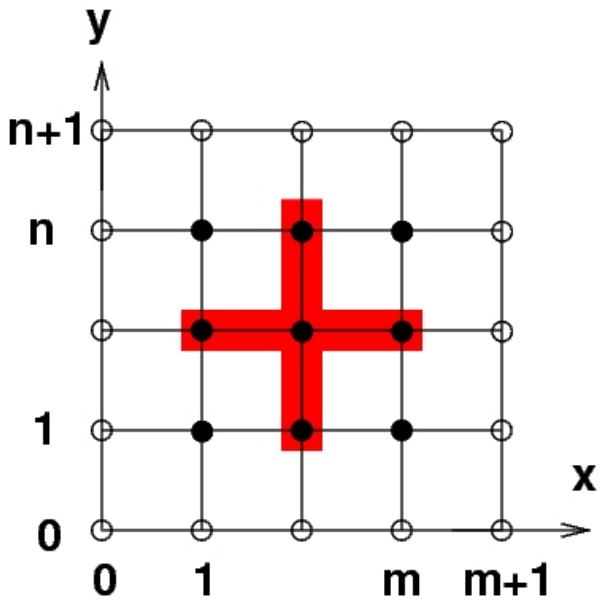
here: $u(x, y) = \sin(\pi\, x) \cdot \exp(\pi\, y)$

- 2nd order central finite-differences, uniform grid:

$$\frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} = 0$$

Jacobi iteration method:

$$u_{i,j}^{n+1} = \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4}$$



Jacobi Iteration of 2D Poisson Problem

Sequential code structure

- initialisation
- iterative loop
  - advance the solution by one step
  - test for convergence
- compute error w.r.t. analytical solution
- write solution to file

$\Rightarrow$ useful to employ modular structure

(sequential source code)

Parallel Code

Individual steps

1. identify concurrency
2. decomposition: (domain/functional)
3. design of communication patterns
4. algorithm implementation
5. validation against sequential results
6. timing/optimization of parallel execution

Concurrency/communication

Symbolic code

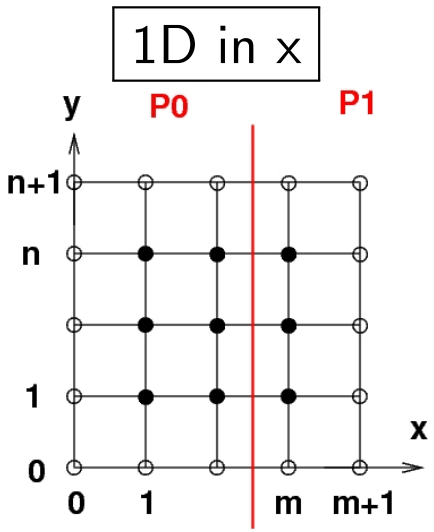
```
(1) initialisation
for each iteration
  (2a) update the solution
  (2b) check for convergence
end
```

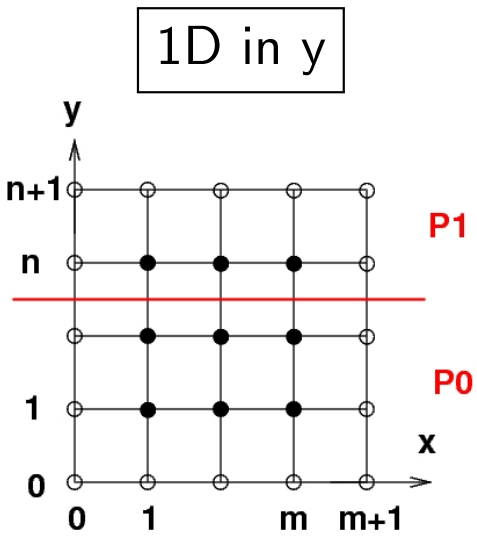
- main work load
- potential for concurrency

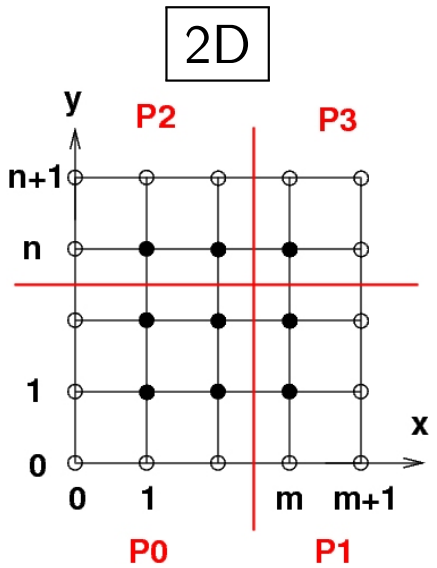
$\Rightarrow$ domain decomposition

starting point: main program template with utilities

Cartesian Domain Decomposition







messages:	$(\mathtt{np}-1)2$	$(\mathtt{np}-1)2$	$(\mathtt{nxp}-1)2\,\mathtt{nyp}$ $+\,(\mathtt{nyp}-1)2\,\mathtt{nxp}$
size:	$(\mathtt{np}-1)2n$	$(\mathtt{np}-1)2m$	$(\mathtt{nxp}-1)2\,n$ $+\,(\mathtt{nyp}-1)2\,m$

- for large np: 2D decomposition $\Rightarrow$ more messages, smaller total volume

Domain/Data Decomposition

- 2D decomposition is most flexible
- BUT: for simplicity we choose 1D decomposition
- no decomposition for first array index

  $\longrightarrow$ contiguous data (FORTRAN)

1D Decomposition in Y-Direction: Pointers

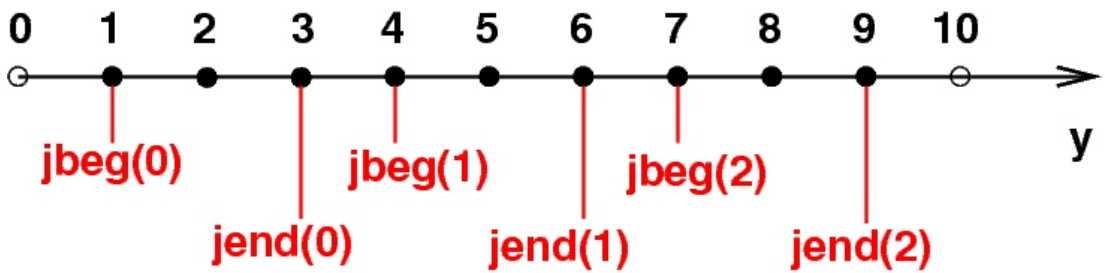
Relation between global and local indices

$\Rightarrow$ subroutine defining pointer arrays (known to all processes):

```
integer jbeg(0:nproc-1)
integer jend(0:nproc-1)
```

example:

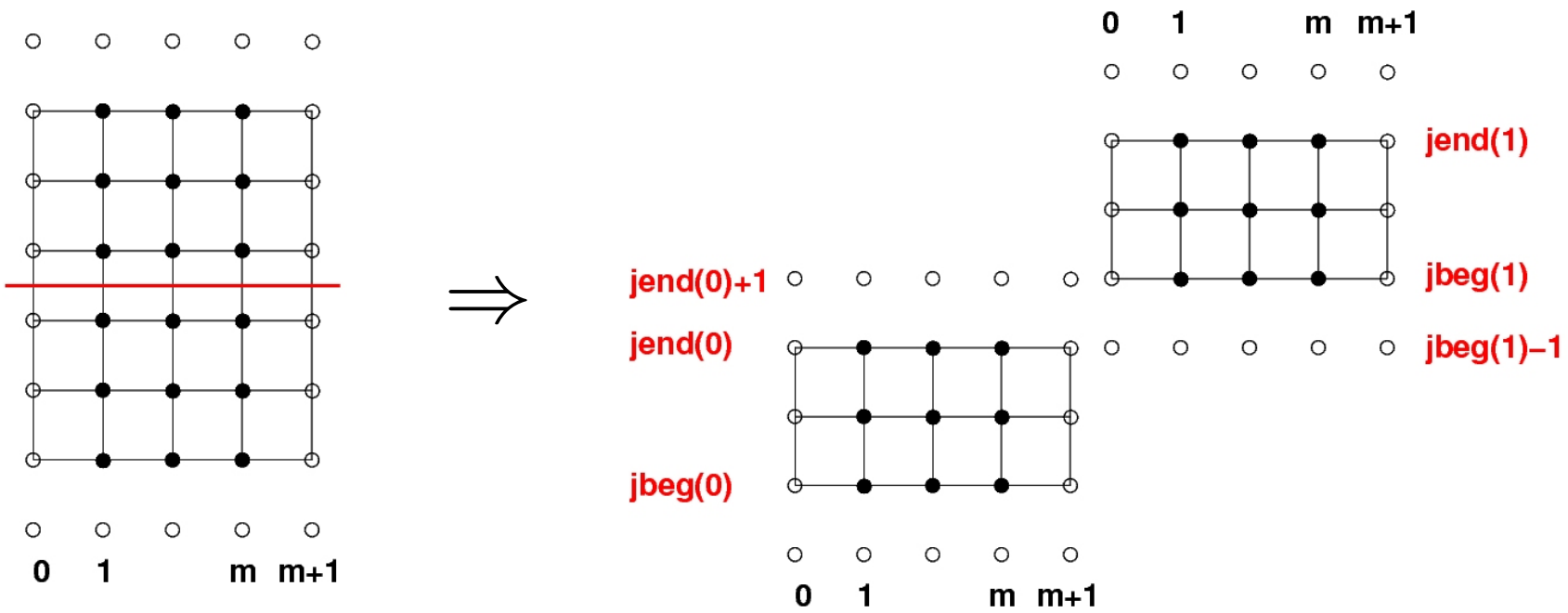
- nproc=3
  n=9



- pointer routine: code template

$\rightarrow$ possible solution

Ghost Cells



$\Rightarrow$ arrays have size: `A(0:m+1,jbeg(myrank)-1:jend(myrank)+1)`

equivalently: `A(0:m+1,1:jend(myrank)-jbeg(myrank)+1)`

Communication

Symbolic code

```
(1) initialisation
for each iteration
  (2a) update the solution
  (2b) check for convergence
toto
end
```

```
(1) initialisation
for each iteration
  (2a) update local part of solution
  (2b) check for convergence
  (2c) exchange ghost-cell data
end
```

Convergence criterion: communication

Compute global maximum of increment

Collective communication

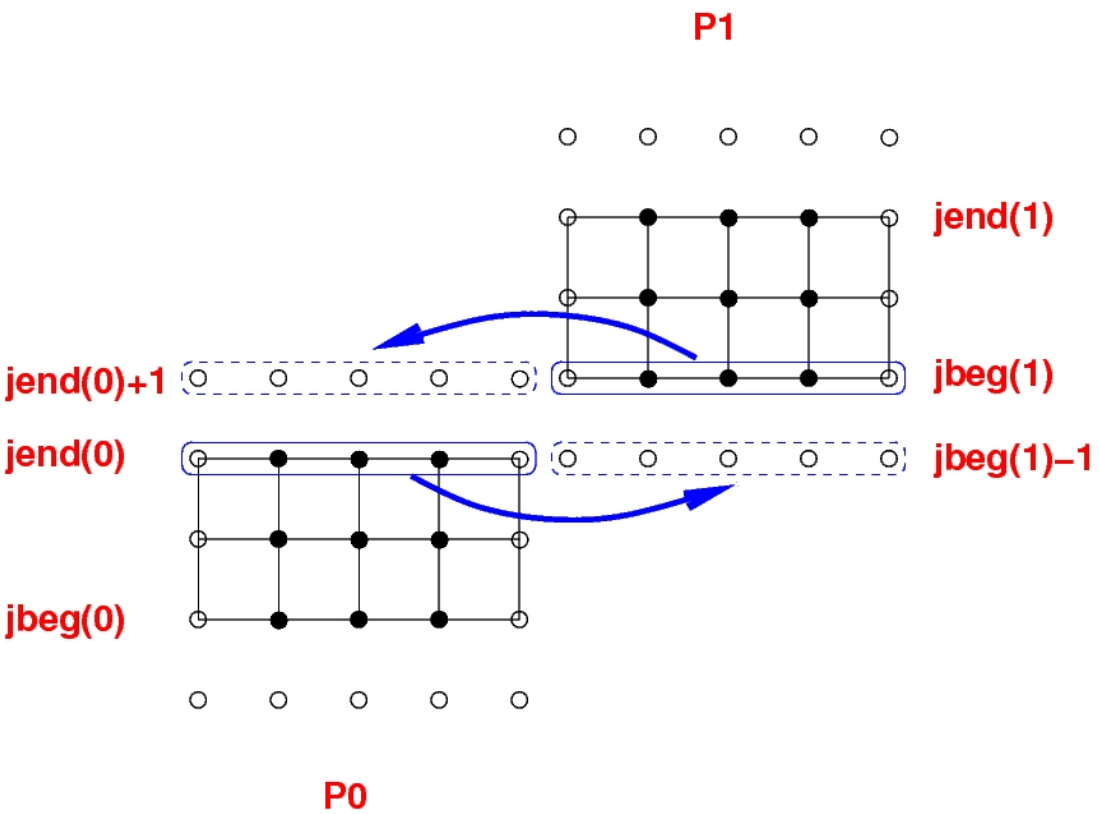
- reduction operation
- `MPI_MAX`
- all-to-all (all tasks need the result)

$\Rightarrow$ `MPI_ALLREDUCE`

possible solution

$\rightsquigarrow$ how could this routine be improved? (latency)

Ghost Cells: Data Exchange



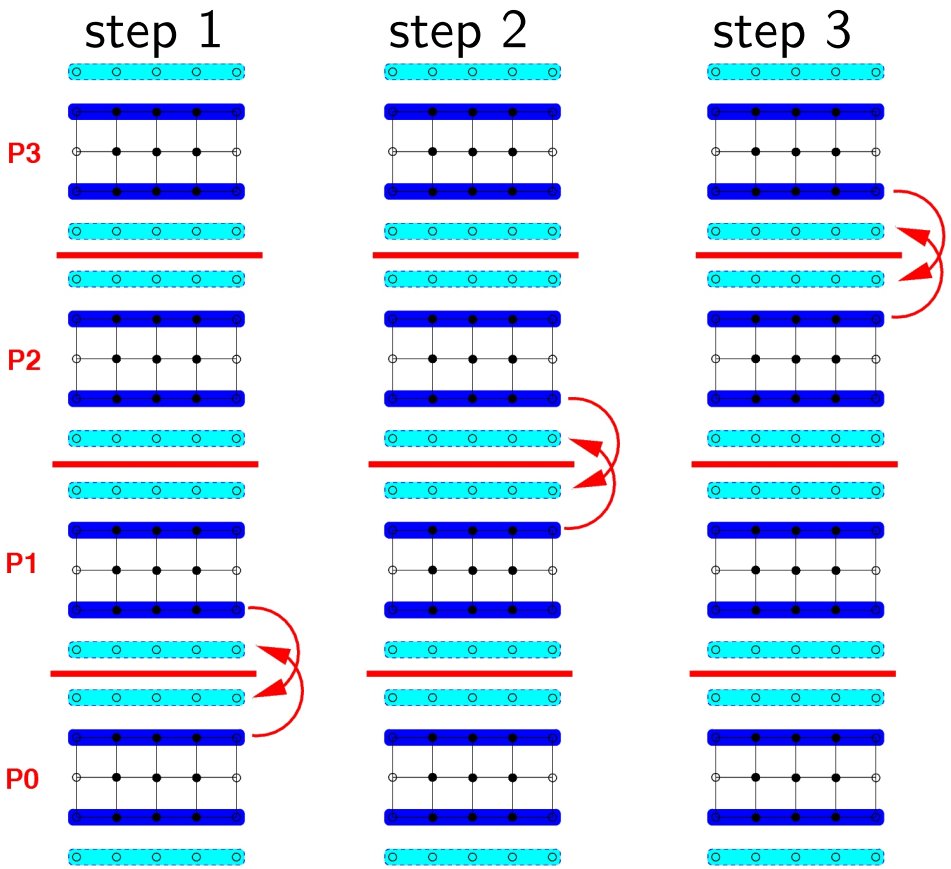
- code template for communication routine

Ghost Cells: Data Exchange

Communication patterns depend on several choices

- blocking vs. non-blocking
- send & recv vs. sendrecv
- order in which communication ops are posted

Communication Pattern 1



- blocking communication
- inferior/superior order

$\longrightarrow$ 'domino' effect:

$\Rightarrow$ needs (`nproc`-1) steps

(source code – Fortran)

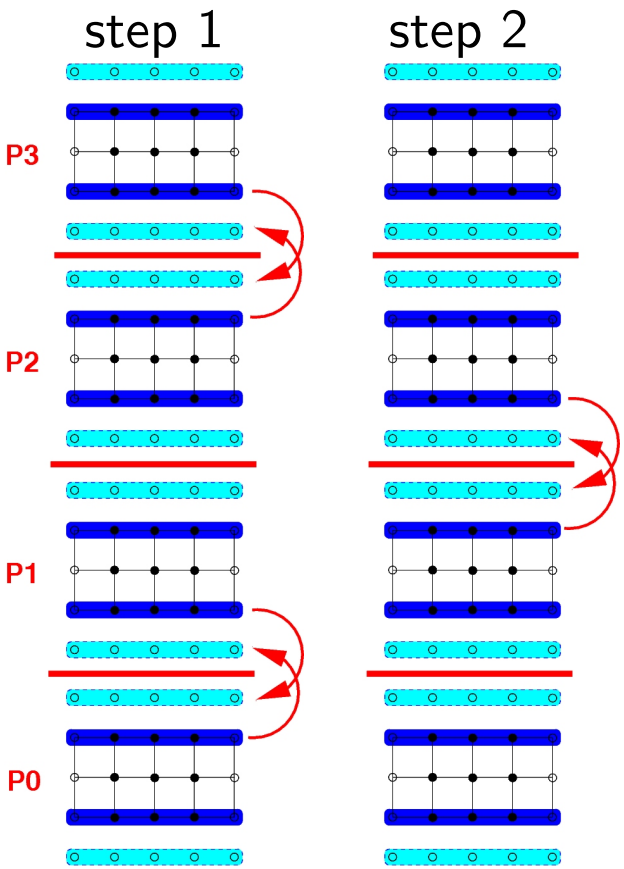
Communication Pattern 2

non-blocking send/receive:

- each processor posts 2 send and 2 receive operations
- MPI internally decides the order of the communication
- all processors wait for completion before proceeding

(source code – Fortran)

Communication Pattern 3



- blocking communication
- odd/even `MPI_SENDRECV`

$\longrightarrow$ alternating pairs communicate

$\Rightarrow$ needs only 2 steps

(source code – C)

Overlapping computation & communication

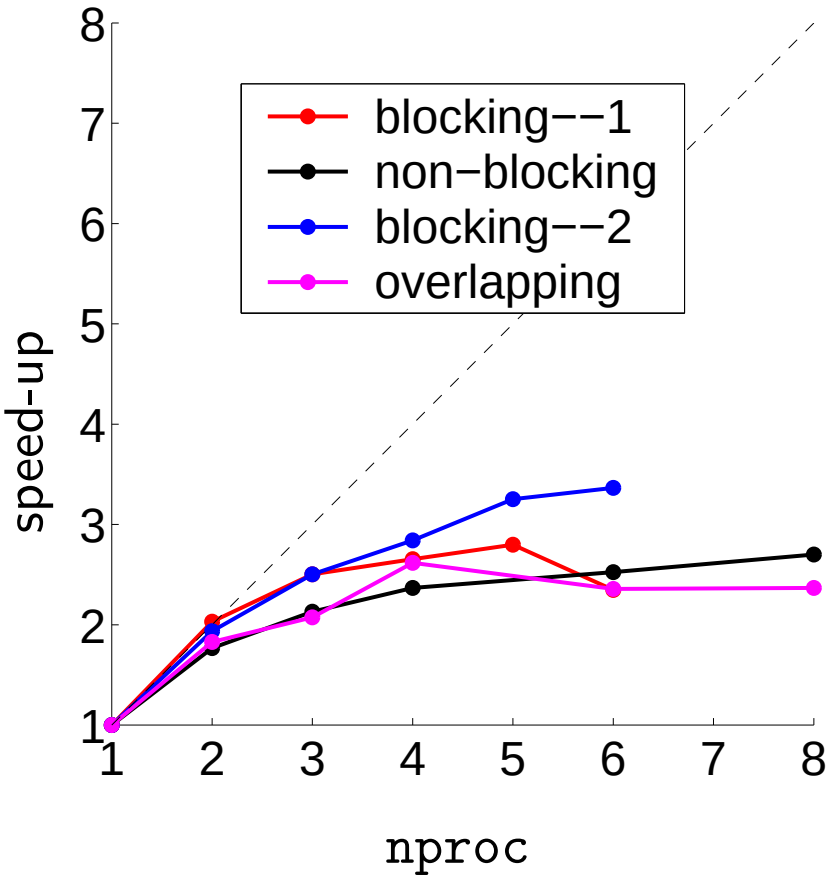
Symbolic code

```
(1) initialisation
for each iteration
  (2a) update local part of solution
  (2b) check for convergence
  (2c) exchange ghost-cell data
end
```

- overlap (2c) with (2a)
- non-blocking communication
- treat grid extrema after interior points, check for termination of communication

(jacobi-step, communication)

Parallel Speed-Up (1)

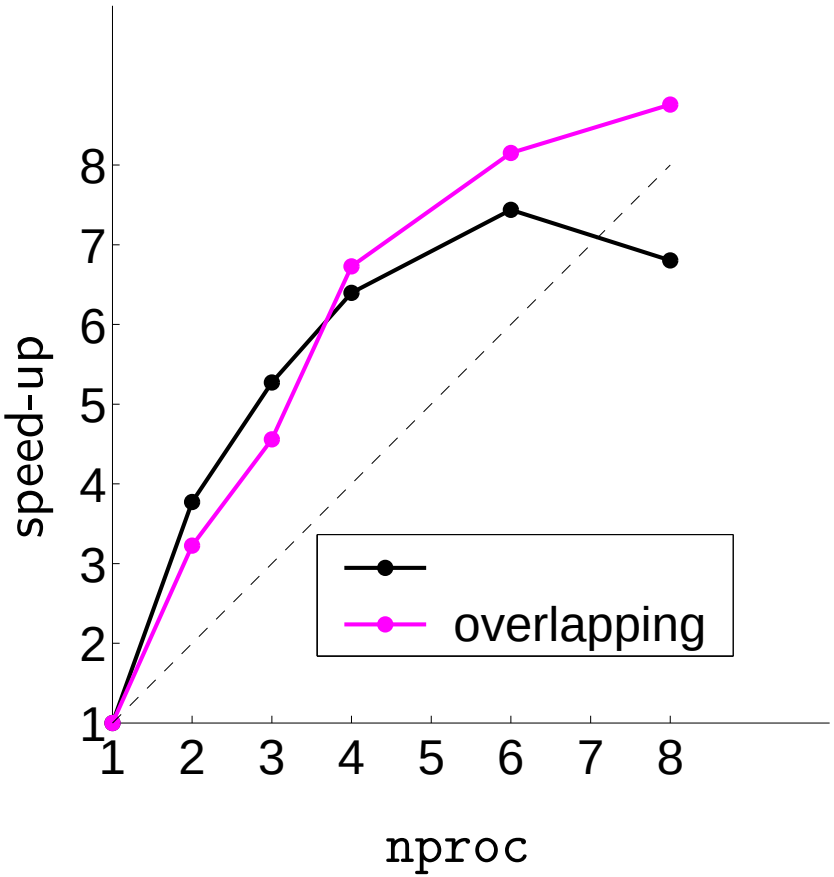


2D Poisson problem:

n=m=250

SGI Altix
shared memory
fenix.ciemat.es

Parallel Speed-Up (2)



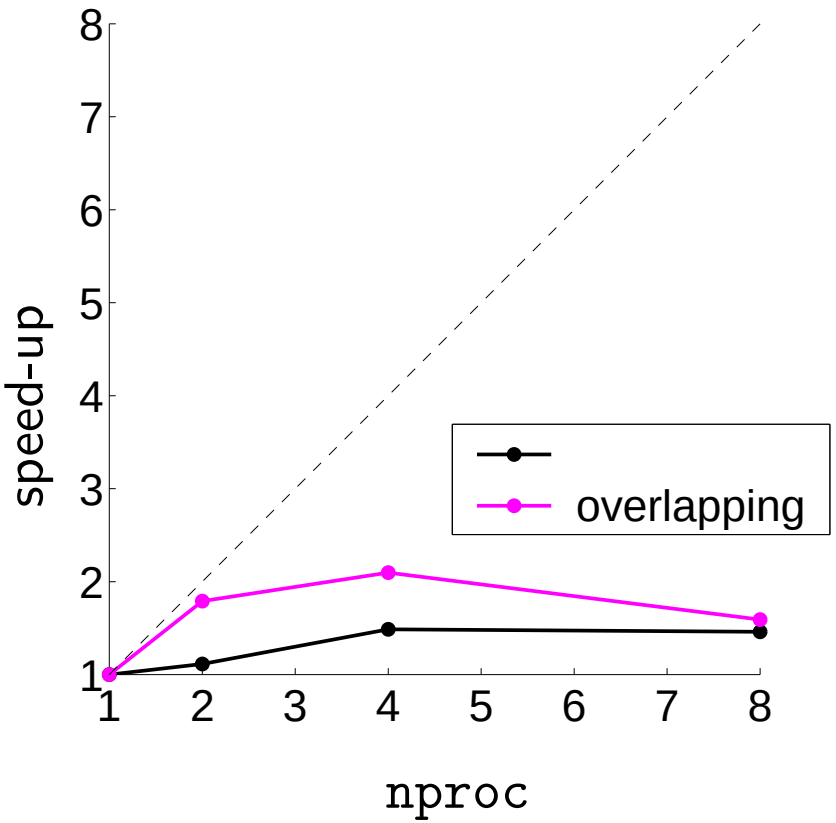
2D Poisson problem:

n=m=500

SGI Altix
shared memory
fenix.ciemat.es

super-linear

Parallel Speed-Up (3)

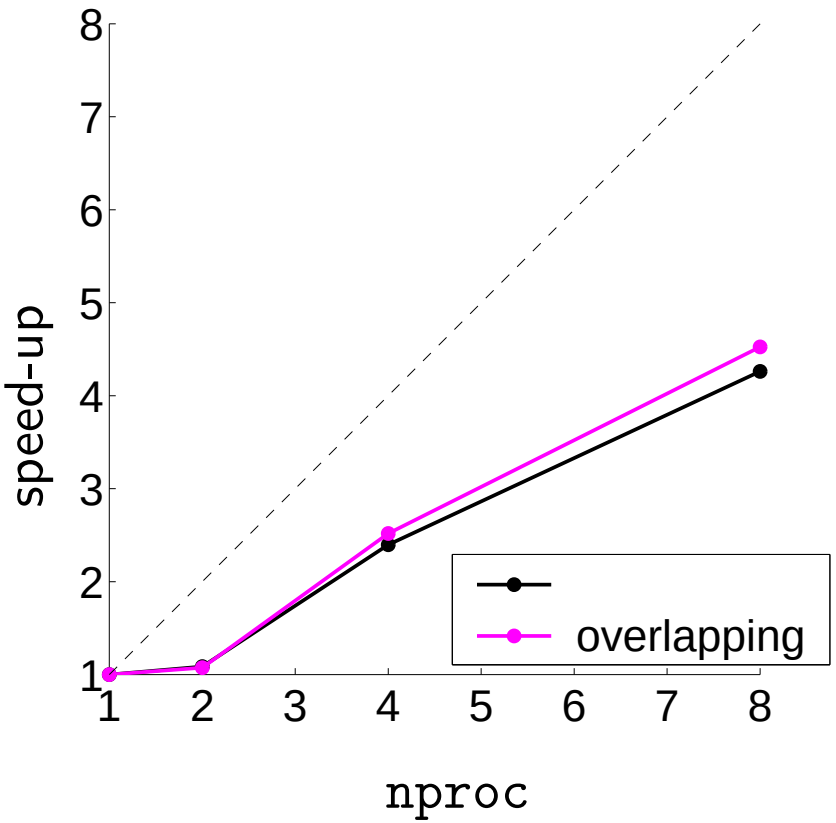


2D Poisson problem:

`n=m=250`

Xeon cluster
Gbit ethernet
`lince.ciemat.es`

Parallel Speed-Up (4)



2D Poisson problem:

`n=m=250`

Xeon cluster
InfiniBand
`lince.ciemat.es`

Parallel performance for 2D Poisson problem

Some observations

- speedup is problem-size dependent
  often: larger size, more parallel work (Amdahl)
- speedup is hardware dependent
  CPU speed; network bandwidth/latency; memory bandwidth
- super-linear speedup due to cache effects
- design of communication patterns matters
- overlapping communication may be beneficial

$\rightsquigarrow$ "optimal" solution differs from machine to machine

Acknowledgement

This lecture heavily draws from the following resource:

- http://webct.ncsa.uiuc.edu:8900
  (Introduction to MPI)

MPI code analysis with MPE

Multi Processing Environment

- library and utility programs distributed with MPICH
- content:
  - routines for creating logfiles
  - graphical visualization tools (jumpshot)
  - shared-display parallel X graphics library
  - miscellaneous other tools

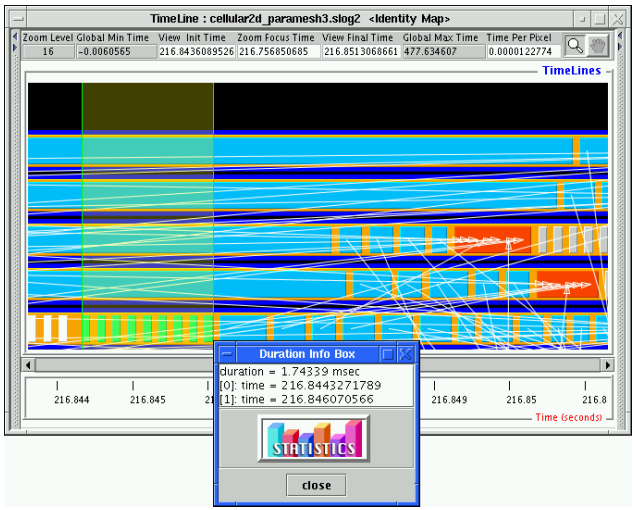
MPI code analysis with MPE (2)

Automatic profiling

- compile code with linker flag `-mpilog`

$\rightarrow$ execution generates logfile `<execname>.clog`

$\Rightarrow$ then visually analyze the result with
`jumpshot <execname>.clog`



Manual profiling example

```
      PROGRAM HELLO  !compile with -lmpe
      INCLUDE 'mpif.h'
      INCLUDE 'mpef.h'
      INTEGER myrank, size, ierr
C     Initialize MPI:
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
      call MPE_INIT_LOG()
      call MPE_DESCRIBE_STATE(1,2,'hello','red:gray')
      call MPE_LOG_EVENT(1,myrank,'write')
      WRITE(*,*)"Hello World!"
      call MPE_LOG_EVENT(2,myrank,'written')
C     Terminate MPI, create log file "mpe_log.clog"
      call MPE_FINISH_LOG("mpe_log")
      call MPI_FINALIZE(ierr)
      END
```

MPI code analysis with MPE

More information

- MPE library documentation
- Performance tool "jumpshot" documentation