Parallelization of the program XBeach

Willem Vermin (SARA)*
Dano Roelvink (UNESCO)†

2008-08-19

Abstract

Here we describe the parallelization of the computer program XBeach. XBeach is a two-dimensional model for wave propagation, long waves and mean flow, sediment transport and morphological changes of the near-shore area, beaches, dunes and back-barrier during storms. It is a public-domain model that has been developed with funding and support by the US Army Corps of Engineers, by a consortium of UNESCO-IHE, Delft Hydraulics, Delft University of Technology and the University of Miami. The typical run times of the program range from hours to days, so it was decided to parallelize the program. The parallelization was done by Willem Vermin at SARA and funded by NCF grant NRG-2007.06

Contents

*willlem@sara.nl
†d.roelvink@unesco-ihe.org

1 Description of the program XBeach

The program is written in Fortran90 and counts circa 12000 non-comment lines, divided over 30 files. The program is reasonably well structured. The relevant data is defined in about 100 arrays (1 to 4 dimensional). A number of subroutines, each with a special function, acts at each time step upon the data. The main subroutines are:

- timestep: determines an appropriate value for the next time step
- wave_bc: wave boundary conditions update
- flow_bc: flow boundary conditions update
- wave_stationary or wave_timestep: to carry out wave time step
- flow_timestep: to carry out flow time step
- transus: to carry out suspended transport time step
- bed_update: to carry out bed level update
- varoutput: to output the desired results

These subroutines are repeatedly called after one another, to modify the arrays describing the state of the system.

2 Description of the parallelization process

2.1 Choice of parallelization paradigm

The program acts on a number of arrays, each describing different aspects of the same rectangular area. In general, to compute a new value of an array element A(i,j), the values of A(i,j), A(i-1,j), A(1,j+1) and so on are needed, along with the corresponding values of other arrays. Therefore a parallelization scheme, involving the distribution of the data among the available processors seems feasible Each processor would compute 'its' piece and communicate the borders with the neighbor processors.

2.2 Reorganizing the program

The program is written in fortran90, but not all features of fortran90 were used. It was still possible to call a subroutine with wrong parameters (type or number), without getting an error message from the compiler. Therefore, all files were modified to generate a module, containing the relevant data and the subroutines. This already uncovered some inconsistencies in the program.

The large number of variables made it difficult to keep the program in an orderly state. The program contains several housekeeping routines such as: the output routine, the allocation and initialization of the distributed data and the creation of debug code. In these parts long lists describing actions on the variables were coded. Therefore it was decided to make this housekeeping more simple and less error prone by creating a "code generating" program: makeincludes.

This method enables the possibility to refer to a variable by the ASCII representation of its name, and to perform actions on all defined variables, without having to know which variables there are. This simplified the code for the output routine considerably, along with the code for the distribution and collection of the data. It makes it also possible to create a routine that checks the consistency of the data for all variables, very useful during debugging.

2.3 Distribution of data

It was decided that the data should be distributed among the processes in two dimensions, as presented below:

0	3	6	9
1	4	7	10
2	5	8	11

This is a 3x4 distribution, using 12 processes. The numbers in the drawing represent the enumeration of the processes, starting with zero (MPI convention). During the computations, process 4 exchanges data with processes 1,3,7 and 5, while process 9 only exchanges data with processes 6 and 10.

2.3.1 Some more details about the distribution of data

Call the global matrix A, dimensioned as A(M,N). Call the sub matrices a0, a1 etc., each dimensioned as a0(m,n), a1(m,n) etc, where m and n can be different

for each matrix a. In the same column the values of n are equal, in the same row the values of m are equal. The matrices overlap, take as example a4:

- the first row contains the same information as the one before the last row of the matrix above
  a4(1,:) == a3(m-1,:)
- the last row (a4(m,:)) contains the same information as the second row of the matrix below
  a4(m,:) == a5(2,:)
- the first column contains the same information as the second last column of the matrix left
  a4(:,1) == a1(:,n-1)
- the last column contains the same information as the one before the last column of the matrix right
  a4(:,n) == a7(:,2)

The matrices on the edges (a0,a1,a3, etc.) do not share their edge(s) which are part of the edges of matrix A with another matrix. The computing domain of a matrix in the middle (a4 for example), shares the elements a4(2:m-1,2:n-1) with A. For matrices on the edges for example the shared elements are a1(2:m-1,1:n-1). On each time step, the elements that are shared with A are computed, and the second and second last columns of the little matrices are communicated with the neighbors. For example the row a3(m-1,:) would be sent to the row a4(1,:). In the program about 100 of these matrices are used, some of them with one or two extra dimensions. However, only a relatively small number of these arrays have to be communicated between processes.

2.4 Parallelization method used

2.4.1 Input and output

The input and output of data is performed by one process: the master process with MPI rank zero. Depending on the properties of the data the following methods are used to distribute the data:

- Broadcast: the data is copied as is to all processes (global variables, parameters of the system, etc.)
- Divide and distribute (all matrices that describe the state of the system in a grid. The appropriate parts of matrix A (see above) are sent to the processes)

Before the master process can output the data, it is collected from all processes.

Using one process for input and output has the advantage that the program will also run on systems where only the master process has the capability to read and write to a file system.

2.4.2 Defining the distribution parameters

At the start of the program, a suitable distribution scheme is determined. Given the number of processes available (P) and the number of gridpoints in x and y direction, the processorgrid is determined such that the total length of the communication edges is minimized. The processorgrid is defined by two integers: MP and NP. In the example above: P=12, MP=3, NP=4. Subsequently, the dimensions of the local matrices (a0, a1, a2 etc.) are determined, such that all these matrices are as equal as possible in size.

2.4.3 Communication subroutines

A number of interface subroutines has been written, tailored to the problem at hand, so that the actual MPI calls are not visible in the program. For example, in stead of coding something like:

```
call MPI_Sendrecv(a(:,2), m, MPI_DOUBLE_PRECISION, &
                  neighbour_left, tag,              &
                  a(:,n), m, MPI_DOUBLE_PRECISION, &
                  neighbour_right, tag,             &
                  MPI_COMM_WORLD,MPI_STATUS_IGNORE,&
                  ierror)
```

one can code as:

```
call xmpi_shift(a,':n')
```

meaning: get the last column of a filled in. This statement suffices to get the last column of all matrices a updated.

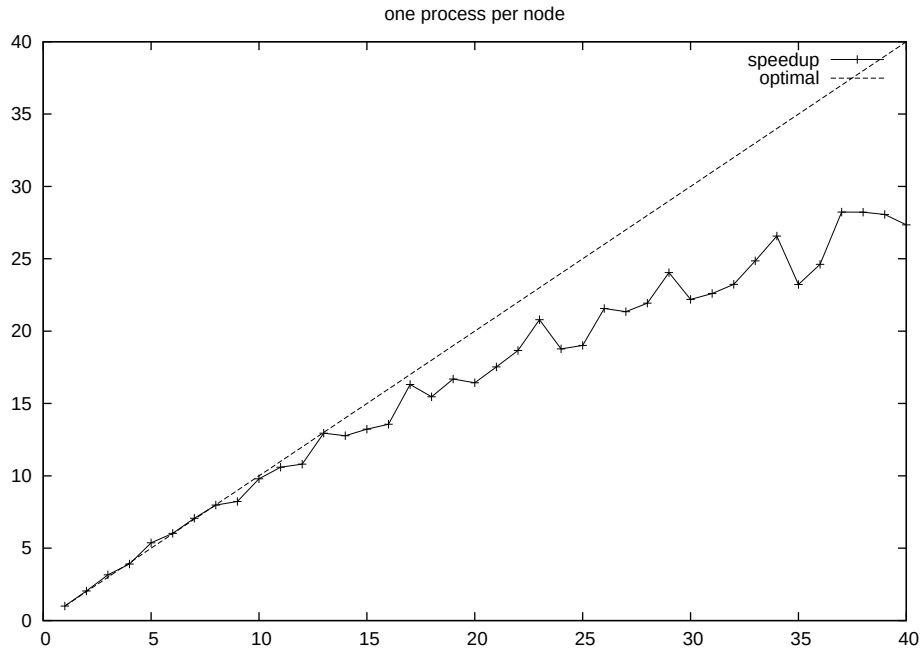
2.4.4 Communication of data

The original program was checked for the places where communication is necessary to maintain consistency. For example, if a matrix is updated, but the border rows and columns are untouched, these rows and columns need to be updated (i.e. received from the neighbors) to maintain consistency. Later in this article we will dive somewhat more in the technical aspects.

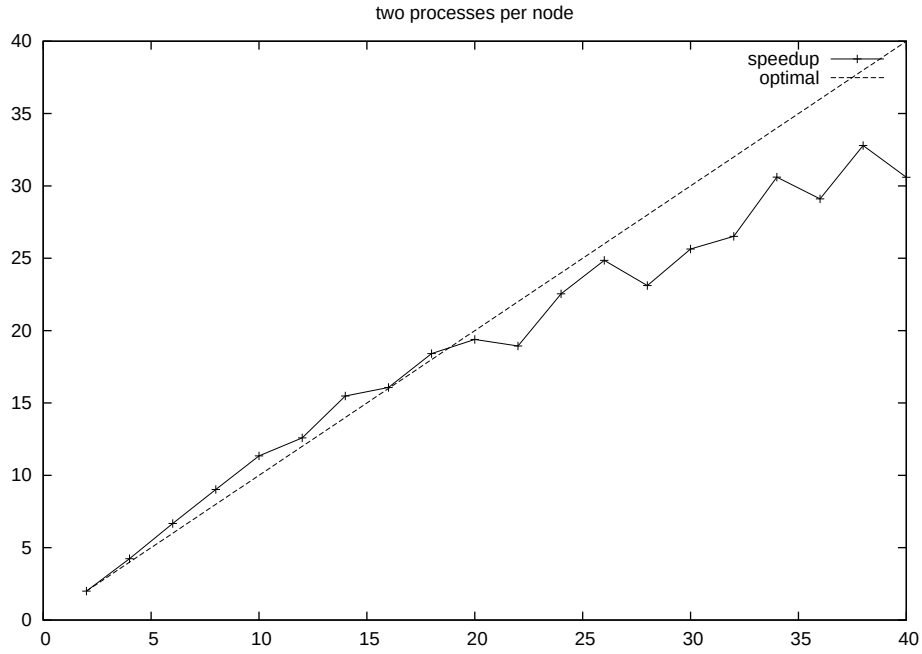
3 Scalability results

The program was executed, using a standard test case: 'humptest.zip'. In this example the size of the global matrices is 101x501. The Lisa system of SARA [1] was used to run the scaling tests. The Lisa system is equipped with an infiniband network between the nodes. The MPI library is OpenMPI-1.2.6 [2] the Fortran90 compiler is gfortran[3]. Computing time of the serial program is about 5 minutes. The number of timesteps is lowered to speed up the scaling measurements and development of the program. In practice, the program would run for several hours.

[1] https://subtrac.sara.nl/userdoc/wiki/lisa/description
[2] http://www.open-mpi.org/
[3] http://gcc.gnu.org/fortran/

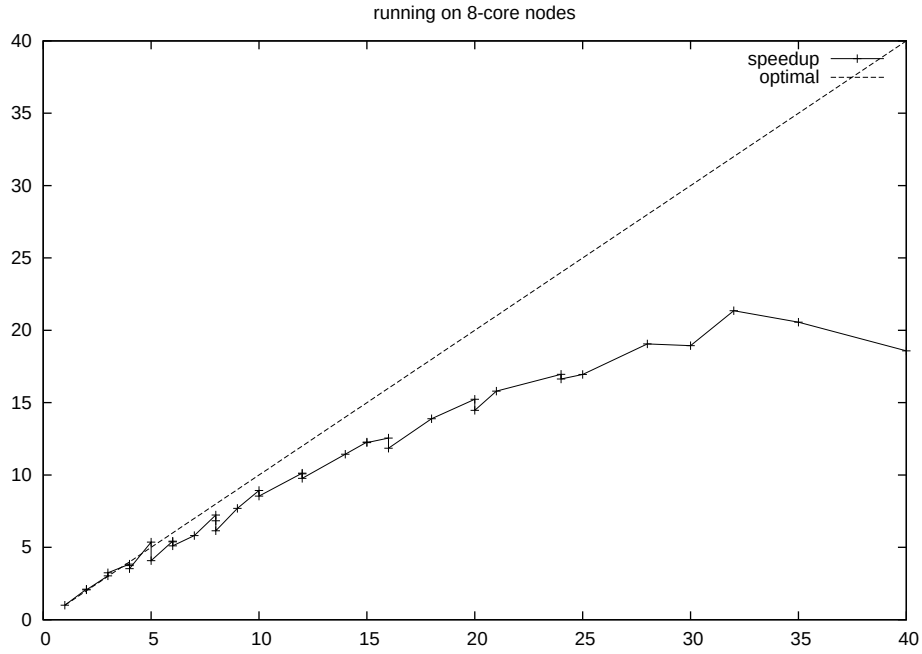


Results using one processor per node. Each node has two one-core processors. The program scales up to 40 processes: he performance using 40 processes is ca. 20 times the performance of the serial version.



Results using 2 one-core processors per node. The speedup-curve is some-

what irregular (probably due to the load variations in the rest of the system: Lisa is very heavily used), but the speedup is about the same as in the above case, using only half the number of nodes.



Results for running the program on nodes equipped with 2 quad-core nodes, making 8 cores per node. Also here we observe a good scaling up to 40 processes (using 5 nodes).

4 Conclusion

This parallelization was successful: usable scalability is 40 processes and probably more. The speedup is about 20 for a quite modest model. Larger models will result in better scalability.

5 Technical details

Here is a detailed description about the parallelization method and the subroutines that were created during the project.

5.1 General conventions

- Code that is only to be executed in the parallel version has to be surrounded as following:

```
#ifdef USEMPI
     call xmpi_shift(s%uu,':1')
     call xmpi_shift(s%uu,'m:')
```

```
#endif
```

- Don't use 'stop' but call halt_program:

```
use xmpi_module
.....
if (error = 1) then
    stop                    ! wrong
    call halt_program       ! good
endif
```

- The following variables are available using xmpi_module

name	meaning	value in serial
xmpi_rank	MPI rank of this process	0
xmpi_size	number of processes	1
xmaster	is this the master process?	.true.
xmpi_isleft	is a(:,1) part of a global border?	.true.
xmpi_isright	is a(:,n) part of a global border?	.true.
xmpi_istop	is a(1,:) part of a global border?	.true.
xmpi_isbot	is a(m,:) part of a global border?	.true.
xmpi_pcol	my column number in processor grid	1
xmpi_prow	my row number in processor grid	1

- Take care that every input/output statement is done on master only, maybe followed by a broadcast.

```
use xmpi_module
.....
   if (xmaster) then
       write(*,*) 'Reading x'
       read *,x
    endif
    call xmpi_bcast(x)
```

- Functions readkey_int and readkey_dbl are MPI-aware, but not readkey:

```
use readkey_module
....
timings  = readkey_int ('params.txt','timings',1,0,1)
....
if(xmaster) then  ! for readkey, test is needed
  call readkey('params.txt','tsglobal',fname)
  open(10,file=fname)
  read(10,*) x
endif
call xmpi_bcast(x)
```

5.2 Parallelization by code spotting

In the program, all relevant matrices are declared as a(1:nx+1,1:ny+1) and have a place in a 'spacepars' derived type. In the serial version there is one

such derived type, in the parallel version there are two: one called 'sglobal' (often abbreviated as 'sg'), the other 'slocal' ('sl'). sglobal is filled in on the master process and has space for all data; slocal contains only the distributed data. sglobal%nx and sglobal%ny are the dimensions of the global grid, whereas slocal%nx and slocal%ny are the dimensions of the local distributed matrices. Using this method there is no need to change anything at the code itself, one only has to take care that data is communicated when appropriate. Below is a table with examples of code patterns and the corresponding actions needed. Here a and b are matrices that are distributed and divided. The dimensions are (1:nx+1,1:ny+1).

pattern	action
`a = .. b ...`	`No action needed`
`a(:,2:ny)= ....`	`call xmpi_shift(a,':1')` `call xmpi_shift(1,':n')`
`a(:,1) = ....`	`call xmpi_shift(a,':1')`
`a(nx+1,:) = ...`	`call xmpi_shift(a,'m:')`
`a(2,:) = ...` `a(3:nx+1,:) = ...`	`This one needs some special attention. In the parallel case, the first line has only meaning in the matricesat the top. The second line is ok for the top matrices, but for the other ones, a(2,:) must be computed in thesame way as a(3,:)`

In general, some actions are necessary when:

- a border column or row gets a special treatment, or is not assigned at all: this is treated with a call to xmpi_shift
- another row or column gets a special treatment. In that case one has to take a good look at the code to find the appropriate action.

One must also take care that the corrections in the border rows and columns are made before they are used. Of course, if one or more of the borders are never actually used in such a way that it influences the output of the program, the corresponding xmpi_shift is not necessary.

5.3 Some MPI related subroutines

The subroutines that interface to MPI are divided in three categories:

- general (general_mpi.F90): this are subroutines that function regardless the environment
- xmpi (xmpi.F90): subroutines that are aware of the parallel environment of the program, they know about the layout of the matrices and know about the communication patterns that are needed
- space (spaceparams.F90): subroutines that know about the data in a spacepars "derived data type"

Some important subroutines and interfaces are listed here. The source code contains instructions how to use them.

file: general_mpi.F90, module: general_mpi_module

name	function
matrix_distr	distributes matrix
vector_distr_send	distributes vector
matrix_coll	collects matrix on master process
decomp	computes optimal division
det_submatrices	determines optimal sizes of matrices
shift_borders	communicates all borders, obsolete

file: xmpi.F90, module: xmpi_module

name	function
xmpi_initialize	initializes MPI
xmpi_finalize	finalizes MPI
halt_program	halts program, serial and parallel
xmpi_determine_processor_grid	determines processor grid
xmpi_bcast	broadcast a variable
xmpi_allreduce	performs an MPI_Allreduce
xmpi_reduce	performs an MPI_Reduce
xmpi_shift	get values for border from neighbour
xmpi_getrow	get a row from another matrix

file: spaceparams.F90, module: spaceparams

name	function
space_consistency	for debugging, checks consistency
space_copy_scalars	copies scalars from and to spacepars
space_distribute_scalars	distribute sclaras in spacepars
space_distribute	distributes matrices
space_shift_borders	communicate all borders, obsolete
space_collect	collects spacepar variables on master
printsum	debugging: prints sum of matrix elements

5.4 Code generation: program makeincludes

In some parts of the program, some kind of bookkeeping is necessary. This can result in boring long pieces of code, difficult to maintain. For example: when the data is read in by the master process, it needs to be distributed among the processes. Since there are more than 100 variables defined in spacepars, at least 100 lines of code would be necessary, and it is all too easy to forget to distribute a newly introduced variable. Therefore a code generating program is developed - "makeincludes" - that reads a simple formatted file, and produces a number of files to be included in the program. The following is now achieved:

- variables in spacepars are defined in one file: the name, the number of dimensions, the dimensions self, and the desired method of distribution:

see the file spaceparams.tmpl. This file also contains a description of the layout desired.

- automatic code generation for the declaration of the spacepars derived type.
- the possibility to get to the value of a variable by using it's name in ASCII (see the example program demo.F90). This proves to be very useful for the subroutine varoutput (varoutput.F90)
- it is easy to write a code that visits all the variables in spacepars, without having to know which variables are available. This was very useful during debugging and finding the cause of inconsistencies that creped in. (See for example subroutine space_consistency in spaceparams.F90)

The program makeincludes generates the following files:

- spacedecl.gen: contains the code needed for the declaration of a spacepars derived type.
- space_alloc_scalars.gen: allocates the simple variables in spacepars. This is necessary, because now all variables in spacepars are declared as pointers.
- space_alloc_arrays.gen: contains the code to allocate the 1,2,3 and 4 dimensional arrays in spacepars.
- mnemonic.gen: defines variables with names like "mnem_E": the variable mnem_E is equal to the string 'E'. Furthermore, an array "mnemonics" is defined with all names.
- indextos.gen: contains code which, given an index, returns a derived type with a pointer to the variable for which mnemonics(index) is equal to the name in ASCII for the variable.
- space_ind.gen, space_inp.gen: they define pointers to the variables in the derived type. Primary goal is to make the code more readable.
- chartoindex.gen: code to convert the name of a variable into an index

Furthermore there are some subroutines defined

- chartoindex (mnemonic.F90): returns the index number of the name given
- indextos (spaceparams.F90): returns a pointer to a variable with a given index number

Program demo.F90 contains an example code to demonstrate how to use this.

5.5 Some notes about Fortran90 and MPI

Using MPI in Fortran90 needs some precautions. This is caused by the way Fortran90 handles arrays when calling a non-Fortran90 subroutine (as is the case with MPI). In Fortran77, the address of the first element of an array is passed, in Fortran90, however, in general a pointer to the first element of a copy of the array is passed. This is necessary because in general it is not possible to tell if the array is contiguous, or a section of another array. For example:

```
subroutine demo(x)
real,dimension(:,:) :: x
call MPI_Bcast(x,size(x),MPI_REAL,0,MPI_COMM_WORLD)
! call MPI_Bcast(x(1,1),size(x),MPI_REAL,0,MPI_COMM_WORLD) ! error
end subroutine demo

program test
real, dimension(100,100) :: y
...
call demo(y(1:100:2,:))
```

In this example, MPI_Bcast will be called with a copy of x, which is no problem: Fortran90 takes care that after the MPI_Bcast the array is copied back. The second MPI_Bcast line would be in error, because the address of x(1,1) would be passed, and MPI_Bcast would broadcast 50x100 elements, contiguous, starting at x(1,1).

In general, Fortran90 will not make a copy if the array is contiguous, and the second MPI_Bcast would be OK if subroutine demo would be called like:

```
call demo(y)
```

So, in general, do not use an array-element as starting address of a buffer (which is common practice in Fortran77), but use the whole array.

Another problem arises with non-blocking sends and receives:

```
subroutine demo(x)
real,dimension(:,:) :: x
call MPI_Isend(x,size(x),MPI_REAL, ...) ! wrong
...
call MPI_Wait(...)
return
end subroutine demo
program test
real,dimension(100,100) :: y
...
call demo(y(1:100:2,:))
```

This will in general give unexpected results. MPI_Isend will get the address of a copy of x, and return while the actual send is still pending. However, after return of MPI_Isend, Fortran90 will free the copy of x, so an invalid buffer (the freed copy of x) will be sent. The same reasoning applies for MPI_Irecv: MPI_Irecv would result in receiving data in an invalid buffer. The actual behaviour of the program is unpredictable.

The solution is to make sure that MPI_Isend is working with the array itself, so one has to make a copy:

```
subroutine demo(x)
real, dimension(:,:) ::x
real, dimension(size(x,1),size(x,2)) :: xx
xx = x
call MPI_Isend(xx,size(xx),MPI_REAL, ...)
...
call MPI_Wait(...)
```

Another problem can exist using the MPI_Scatterv and the like subroutines. These subroutines expect a description of the layout of the data (the so-called counts and displacements arrays). Also in this case it is important to make sure that the data is contiguous, otherwise the displacements can be invalid.

5.6 Compilation and running

The parallel code is developed on systems running Linux. Here follows a description how to compile and run the program. On Windows systems, the details can differ.

5.6.1 Compilation

A Makefile is provided, in principle compilation is as easy as:

```
USEMPI=yes make install # produce a parallel version:
                        # xbeach.mpi
make clean
make install            # produce a serial version:
                        # xbeach
```

to get the parallel and serial versions. They are installed in the directory ../bin . Important macro's are:

name	function
USEMPI	when defined: generate parallel program
USEMPE	when defined: produce trace files for jumpshot
F90	the fortran compiler to use

The Makefile is pretty simple, it should not be difficult to adapt to a local situation. Do not define USEMPE for a production version of the program.

When compiling Fortran90 files, it is important to have correct dependencies, especially when module files are generated (as is the case here) and when one wants to run make in parallel (make -j) to speed up the compilation process. Therefore a simple script has been made, makedepo, which takes care of dependencies. It is called by the Makefile when no file named "DEPENDENCIES" is present. One can force a re-generation of this file by

```
make dep
```

Other things one can make:

```
make clean       # gets rid of .o and .mod files
                 # and test programs
make realclean   # gets rid of everything except
                 # files needed for compilation
```

```
make testgenmodule # make program testgenmodule
                   # that tests the communicating
                   # subroutines.
                   # USEMPI must be defined
make demo        # make program demo
```

Furthermore, a script 'maketags' is provided, which produces a "tags"[4] file, very useful in combination with the vim or emacs program editors.

5.6.2 Running the program

We give two examples, one for OpenMPI, one for MPICH2, to run the program on 8 processes:

OpenMPI

```
mpiexec -n 8 directory-to-bin/xbeach.mpi
```

MPICH2

```
mpdboot [-n number-of-nodes -f file-with-node-names]
mpiexec -np 8 directory-to-bin/xbeach.mpi
```

[4] exuberant tags: http://ctags.sourceforge.net/