**Institute for System Programming,
Russian Academy of Sciences**

# mpC programming environment

# User's Guide

# version 2.0.3

**Moscow, 1999**

# 1. Definition of terms

The following terms are used in this document:

- Computing space (a language term) - a set of typed virtual processors of different performances connected with links of different bandwidths;
- Distributed memory machine (an implementation term) - any computing system running MPI (for example, cluster of workstations, heterogeneous network of workstations and/or PCs, a specialized parallel computer, or everything taken together);
- Virtual parallel machine (an implementation term) - a set of processes representing virtual processors of the computing space and running over distributed memory machine.
- Host workstation (an implementation term) - any workstation or PC that may be used as a working place of the user. It is intended that the user may start running mpC applications only from host workstations.

# 2. Outline of the mpC programming environment

Currently, the mpC programming environment includes a compiler, a run-time support system (RTS), a library, and a command-line user interface. All these components are written in ANSI C.

The compiler translates an mpC program into a ANSI C program with calls to functions of RTS. The compilation unit is a source mpC file. The compiler uses optionally either the SPMD model of target code, when all processes constituting a target message-passing program run the identical code, or a quasi-SPMD model, when it translates the source mpC file into 2 distinct target files: the first for the virtual host-processor and the second for the rest of virtual processors.

RTS manages the computing space and provides any necessary communications. RTS encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of compiler components.

The library consists of a number of functions which provide low-level efficient facilities as well as support debugging mpC programs.

The user interface consists of a number of programs supporting the creation of a virtual parallel machine and monitoring the execution of mpC applications on the machine. While creating the machine, its topology is detected by a topology detector and saved in a file used by RTS. The topology detector executes a special benchmark to detect performances of workstations constituting the target distributed memory machines, the number of processors in each of these workstations, as well as bandwidths of links connecting the workstations (optionally).

All processes constituting the target program are divided into 2 groups - the special process named *dispatcher* playing the role of the computing space manager, and general processes named *nodes* playing the role of virtual processors of the computing space. The dispatcher works as a server accepting requests from virtual processors. The dispatcher does not belong to the computing space.

In the target program, every network or subnetwork of the source mpC program is represented by a set of nodes called *region*. So, at any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them is the responsibility of the dispatcher. The only exception is the pre-hired host-node representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

If a region represents a network, creation of the region involves the parent node, the dispatcher and all free nodes. The parent node sends creation request containing the necessary information about the network topology to the dispatcher. Based on this information and the information about the topology of the virtual parallel machine, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. Deallocation of network region involves all its members as well as the set of free nodes and the dispatcher.

If the region represents a subnetwork, its creation involves only members of the enclosing region. Deallocation of subnetwork region involves only its members.

The dispatcher keeps a queue of creation requests that cannot be satisfied immediately but can be served in the future. It implements some strategy of serving the requests aimed at minimization of the probability of occurring a deadlock. The dispatcher detects such a situation when the sum of the number of free nodes and the number of such hired nodes that could be released is less than the minimum number of free nodes required by a request in the queue. In this case, it terminates the program abnormally.

# 3. Supported systems

We tried to write all the components of the mpC programming environment in such a way to avoid any problem with its installation on any Unix system having C compiler supporting ANSI C. We have checked it for the following platforms:

- Sun workstations running Solaris 2.3/2.4/2.5 or SunOS 4.1.3 with gcc versions 2.6.3, 2.7.0, 2.7.2 and SPARCworks Professional C 3.01;
- HP9000 workstations running HP-UX 9.07 with gcc version 2.7.2 and c89;
- PC running Lunix 4.0 with gcc version 2.7.2.
- DEC Alpha running OSF1 V3.2.

We tried to write the mpC compiler in such a way to avoid any problem with compilation of generated code on any Unix system having C compiler supporting ANSI C. We have checked it for the platforms listed above.

We tried to write RTS in such a way to ensure its correct work for any implementation of MPI supporting full MPI 1.1 standard as an underlying comunication platform. We have checked it for LAM MPI versions 5.2, 6.0, 6.1 (for the platforms listed above) and for MPICH version 1.0.13 (for Sun workstations running Solaris and HP9000 workstations running HP-UX 9.07).

The current version of the command-line user interface is written in such a way to work correctly for two implementations of MPI - LAM and MPICH. We have checked it for LAM versions 5.2, 6.0, 6.1 (for the platforms listed above) and for MPICH version 1.0.13 (for Sun workstations running Solaris and HP9000 workstations running HP-UX 9.07).

# 4. mpC compiler

To call the mpC compiler one should type:

```
mpcc [options] filename
```

**mpcc** processes an input file through one or more of tree stages: preprocessing, analysis, and generating one or two C files. For preprocessing we use a standard preprocessor and recommend to use GNU `cpp`. Only one input file may be processed at once. The suffix '.mpc' is used for mpC source files, and the suffix '.c' is used for processed mpC files. **mpcc** puts output C files into the current directory.

## 4.1 Options.

All options must be separated. For example, '-hetmacro' is quite different from '-het -macro'. All options different from described bellow are considered as options of the preprocessor.

-E

Stop after the preprocessing stage; do not run the compiler proper. The output is preprocessed source code, which is sent to standard output.

-analyse

Compiler provides parsing and semantical analysis. C files will not be generated.

-k*mode*

Selects one of four parser modes. *mode* may be SHORT, ANSI, LONG, and ALL. By default the SHORT mode is used. This mode allows to use only the short form of mpC keywords. The ANSI mode allows to use only ANSI C keywords. The LONG mode allows to use only the full form of mpC keywords. The ALL mode allows to use both forms of mpC keywords. For example,'net' and 'mpc_net' are identifiers in the ANSI mode and mpC keywords in the ALL mode. 'net' is an identifier in the LONG mode and an mpC keyword in the SHORT mode. Finally, 'mpc_net' is an identifier in the SHORT mode and an mpC keyword in the LONG mode. Presence of these modes supports the compatibility with previously written C code.

-macro

Forbids to use some macros in generated C files. The macros contain parameterized code standing for long pieces of code. Code with macros is shorter but may be less undestandable.

-out

Directs output of **mpcc** to standard output instead of C file.

-het

Makes compiler produce two output C files. By default, for source mpC file 'name.mpc' **mpcc** produces one output C file 'name.c'. If -het is typed, then **mpcc** will produce two output files: 'name_host.c' containing code for the virtual host-processor and 'name_node.c' containing code for the rest of virtual processors. This option allows to isolate code with input/output operations to a single processor.

## 4.2 Pragmas.

A #pragma directive of the form

#pragma keywords *mode*:

is supported by **mpcc**. This pragma has the same affect on mpC keywords as option -k described above and allows one to use the same header files in C and mpC sources.

# 5. How to start up

By now, we dealt with local networks of workstations running UNIX (including PCs running LINUX) as a DMM. Any workstation that may be used as a working place of the user is called a host workstation. It is intended that the user may start running mpC applications only from host workstations.

To start working with the mpC environment, the user must have it installed on each of workstations constituting his DMM (see Installation Guide).

Then the user should become an authorized user with the same name on each of workstations constituting the DMM.

Then the user should make sure that on each of these workstations in his home directory file '.rhosts' exists and contains names of **all** workstations constituting the DMM.

Then on each of these workstations the user should modify corresponding files (for example, '.cshrc' if he uses C shell) in his home directory to determine environmental variables WHICHMPI, MPIDIR, MPCHOME, MPCTOPO, and MPCLOAD of his shell.

**Notes.** Sometimes one needs modify different files for local and remote invocation of shell. For example, for PC running Linux 4.0 the user should modify files '.bashrc' and 'bash_profile' if he uses Bourne shell.

There are a small number of restrictions dependent on the used value of WHICHMPI:

- When using LAM, it may be needed to determine environmental variable TROLLIUSHOME setting it to the same value as MPIDIR.
- When using MPICH, environmental variable MPCHOME must be set to the same value on all workstations constituting the DMM. To ensure it, the user may need to use the Unix **ln** command to make necessary hard or soft links.
- When using MPICH, environmental variable MPCLOAD must be set to the same value on all workstations constituting the DMM. To ensure it, the user may need to use the Unix **ln** command to make necessary hard or soft links.
- When using MPICH, the user should make sure that he has write access to directory $MPIDIR/bin/machines (equally, $MPIDIR/util/machines) on each of host workstations.

Then on each workstation the user should create his own directories $MPCTOPO, $MPCTOPO/log, and $MPCLOAD. No two workstations or users can share these directories. The user should make sure that he has write access to these directories.

Then on each workstation the user should modify corresponding files in his home directory to add directories $MPIDIR/bin, $MPIDIR/lib, $MPCHOME/bin, $MPCHOME/lib and $MPCLOAD to his PATH. To avoid name conflicts, make directory $MPCLOAD first in the search path.

In addition, the user should add directories $MPIDIR/lib and $MPCHOME/lib to his ld path (by changing LD_LIBRARY_PATH for Solaris, LPATH for HP-UX and so on).

# 6. Virtual parallel machine

The next step is the description of the virtual parallel machine (VPM) which will execute mpC applications. The description is provided by a manually-written VPM description file which should be placed to the $MPCTOPO directory. The name of this file is just considered as a name of the described VPM.

## 6.1 Virtual parallel machine description file.

A VPM description file consists of lines of two kinds. Lines starting with symbol '#' are treated as comments. All other lines should be of the following format:

```
<name> <number_of_processes>
```

where `<name>` is the name of the corresponding workstation as it appears in the system '`/etc/hosts`' file, and `<number_of_processes>` is the number of processes to run on the workstation. The host workstation must go first in the file. The virtual host-processor will be mapped to a process running on this workstation.

For example, the following file describes VPM which runs on DMM consisting of three workstations (`alpha`, `beta`, and `gamma`), five processes running on each workstation, and the host workstation is `alpha`:

```
# three workstation each running 5 processes
alpha 5
beta 5
gamma 5
```

The following example describes VPM with the same total number of the processes, but running on the single workstation `alpha`. It may be useful for debugging mpC applications:

```
# simple topology for debugging
alpha 15
```

The actual total number of running processes is greater then the number specified in the description file. A process for the dispatcher is added automatically and runs on the host workstation. The virtual host-processor is always placed on the host workstation.

# 7. Environmental variables

## 7.1 WHICHMPI.

Currently, $WHICHMPI should be

- LAM, if you use a LAM implementation of MPI;
- MPICH_P4, if you use a MPICH implementation of MPI configured with the ch_p4 communications device;
- MPICH, if you use a MPICH implementation of MPI configured with any valid communications device not having to be ch_p4.

WHICHMPI should be set to the proper value on host workstations.

## 7.2 MPIDIR.

$MPIDIR is a directory where MPI has been installed. MPIDIR should be set to the proper value on each workstation of DMM.

## 7.3 MPCHOME.

$MPCHOME is a directory where the mpC programming environment has been installed. MPCHOME should be set to the proper value on each workstation of DMM.

Subdirectory $MPCHOME/bin holds all executables and scripts of the mpC programming environment.

Subdirectory $MPCHOME/h holds all specific mpC header files as well as header ‘mpc.h’ containing declarations of the mpC library and embedded functions.

Subdirectory $MPCHOME/lib holds RTS object files ‘mpcrts.o’ and ‘mpctopo.o’.

With WHICHMPI set to MPICH, the user should ensure MPCHOME to have the same value on all workstations of the DMM. If mpC has been installed in different directories on different workstations, you can use the Unix **ln** command to make necessary hard or soft links and ensure the property.

## 7.4 MPCLOAD.

$MPCLOAD is a directory for C files, object files, libraries and executables related to user’s applications. MPCLOAD should be set to a proper value on each workstation of DMM. No two workstations or users can share the directory. The user should have write access to the directory.

With WHICHMPI set MPICH, the user should ensure MPCLOAD to have the same value on all workstations of the DMM. In particular, you can use the Unix **ln** command to make necessary hard or soft links and ensure the property.

## 7.5 MPCTOPO.

$MPCTOPO is a directory for VPM description files as well as all topological files produced by the mpC programming environment. MPCTOPO should be set to a proper value on each workstation of DMM. The mpC programming environment saves a file specifying the current VPM in subdirectory $MPCTOPO/log. No two workstations or users can share these directories. The user should have write access to these directories.

# 8. How to run mpC applications

To run an mpC application on a described virtual parallel machine, the user should proceed the following steps:

1. create the necessary VPM by the **mpccreate** command. Immediately after that, the VPM is opened;
2. if the necessary VPM has been created earlier, open it by the **mpcopen** command instead of its creation;
3. put all \`.c' and \`.o' user's files, necessary to produce executable file, into the $MPCLOAD directory on the host workstation;
4. broadcast all the files, necessary to produce executable, from the $MPCLOAD directory on the host workstation to $MPCLOAD directories on other workstations constituting the DMM by the **mpcbcast** command;
5. create an executable file on each of workstations constituting the DMM by the **mpcload** command;
6. run the executables by the **mpcrun** command.

Additionally,

**mpctouch** displays status of the VPM and all its processes.

**mpcclean** cleans VPM.

**mpcclose** ends the work with the current VPM.

**mpcmach** prints name of the current VPM.

For simple examples of the session of the work with the mpC programming environment see section 10.

## 8.1 mpccreate.

```
mpccreate name
```

where name is the name of the VPM to create. The command uses the $MPCTOPO/name description file. The command creates VPM, i.e. produces all necessary files for it.

In particular, the **mpccreate** command creates the $MPCTOPO/`name`.topo file, containing a description of the topology of the created VPM and used by RTS in run time. Currently, the file consists of pairs of lines of the form:

```
# <name_of_workstation>
s<number_of_processors> p<performance> n<number_of_processes>
```

where `<name_of_workstation>` is the name of the corresponding workstation as it appears in the $MPCTOPO/name description file, `<number_of_processors>` is the number of physical processors in the workstation, `<performance>` is an integer number characterizing the performance of each of these physical processors and `<number_of_processes>` is the number of processes running on the workstation. We recommend to check out the file after creation of the VPM, since the detected topological characteristics can be rough enough if the background workload of the corresponding DMM was essential and uneven during the work of the **mpccreate** command.

Once created, the VPM is accessible to be opened by **mpcopen**. Note that **mpccreate** is expensive and executes a lot of computations and communications, so it may take a few minutes to create new VPM.

## 8.2 mpcopen.

```
mpcopen name
```

where `name` is the name of the VPM to open. The VPM must be created earlier. After opening, the VPM is accessible for **mpcbcast**, **mpcload**, **mpcrun**, **mpcclean**, **mpctouch**, **mpcmach**, and **mpcclose**.

## 8.3 mpcbcast.

```
mpcbcast [file1 file2 ... ]
```

The command broadcasts files listed from directory $MPCLOAD on user's workstation to directory $MPCLOAD on the rest of workstations constituting DMM. Only file names without paths must be typed.

## 8.4 mpcload.

```
mpcload [-het] -o target [file1.c file2.c ... ] [file01.o
file02.o ... ]  [file11.a file12.a ... ]
[options_to_all_nodes] [-host] [options_to_host_only]
```

The **mpcload** command produces executable `target` from in directory $MPCLOAD on each of workstations constituting the DMM. Do not use a path in `target`.

The command produces the executable from `.c` , `.o` and `.a` files. The name of each of these files either uses no path or uses the full path. The first case means that the file is searched in directory $MPCLOAD.

Option `-het` must be used if workstations participating in the VPM are not binary compatible. The user may use the option even if all the workstations are binary compatible.

Option `-host` separates options necessary to all nodes and options necessary only for the virtual host-processor. In addition, if this option appears then:

- `target` for the virtual host-processor is produced from fully-named files and shortly-named files, whose names are produced from names of shortly-named `.c` , `.o` and `.a` files as they are typed in the command line by addition `_host` to the end of name;
- `target` for other nodes is produced from fully-named files and shortly-named files, whose names are produced from names of shortly-named `.c` , `.o` and `.a` files as the are typed in the command line by addition `_node` to the end of name.

`Options_to_all_nodes` and `options_to_host_only` are may be any proper C compiler options. Note, that if option `-c` is used, and hence `target` is an `.o` file, then option `-host` can not be used.

## 8.5 mpcrun.

```
mpcrun target [-- params]
```

The command runs mpC application `target` on the current VPM and passes parameters `params` to this application. Do not use a path in `target`.

## 8.6 mpctouch.

```
mpctouch [-p]
```

The command checks status of the current VPM and displays it. The VPM may be ready or busy. If option `-p` is typed then the status of all nodes is displayed. Currently, the command makes sense only for LAM implementation.

### 8.7 mpcclose.

```
mpcclose
```

Closes the current VPM.

### 8.8 mpcclean.

```
mpcclean
```

The command cleans the current VPM and makes it ready to run new mpC application. The command should be used in case of abnormal termination of the previous command or mpC application. Currently, the command makes sense only for LAM implementation.

### 8.9 mpcmach.

```
mpcmach
```

Prints the name of the current VPM.

### 8.10 mpcdel.

`mpcdel` *`name`*

where *`name`* is the name of a VPM. The command deletes the VPM (that is, deletes all system files related to the VPM).

# 9. Debugging mpC applications recommendations

Debugging an mpC application isn't yet an easy task, but it is much simpler than debugging an arbitrary MPI application, because of absence of nondeterminism.

There are at least three levels of debugging.

At the top level we suggest to include in mpC code calls to MPC_Global_barrier() and MPC_Barrier() to split program execution into small debuggable portions. It may be helpful to use the MPC_Printf() function to output node coordinates, and values of variables. But there are no guarantee you to see all MPC_Printf messages, because some errors make message-passing subsystem failed. It is also possible to use ‘printf’, but part of output done on remote computers will be lost. However, we strongly recommend to start debugging using a single workstation as DMM. All error messages include either position in the mpC source file or ‘0,0’, if the error takes place in the dispatcher process.

The middle level includes including printf's and barriers in C code generated by mpcc. It is a bit more sophisticated than previous approach, because the user needs to understand the logic of the generated C code and RTS kernel calls. If the user sets environmental variable MPC-TRACEMAPFILE to an absolute name of file, then in the corresponding file he will obtain a table, where each pair of lines contains info about network allocation. Sign “+” stands for previously allocated process, sign “-” stands for currently unemployed one, and “p” - for the parent node of the network. In the second line user can see node ranks in the created network.

See the following example:

| 0<br>p | 1<br>- | 2<br>- | 3<br>- | 4<br>- | 5<br>- | 6<br>- | 7<br>- | 8<br>- | 9<br>- | 10<br>- | 11<br>- |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | - | - | - | - | - | - | - | - | - | - |
| p | - | - | - | - | - | - | - | - | - | - | - |
| 0 | - | - | 1 | - | - | - | - | - | - | - | - |
| p | - | - | + | - | - | - | - | - | - | - | - |
| 0 | - | - | + | - | - | - | - | - | - | - | - |

Three networks were created. The first network contains only one node. It is the host. The second network contains nodes 0 (host) and 3, and its parent is the host. The third network is similar to the first one, but at the moment, when it was created, node 3 was already used.

At the low level of debugging, the user may turn kernel tracing on and use MPI utilities such as ‘state’ and ‘mpitask’ (LAM 6.0). To turn tracing on, the user needs to close the current virtual parallel machine, then set the MPC_DEBUG environmental variable to 1 or 2 and reopen the machine.

To debug an application, which hangs without error messages, first call ‘mpitask’ to find processes which do not respond. In many cases they “die” due a simply “C error”, such as an uninitialized variable, dividing by zero and so on. The trace is useful for finding the point in the code where the disaster occurs. When all processes are alive, ‘mpitask’ shows operations, where the processes are blocked.

# 10. Examples of mpC sessions

## 10.1 Simplest example.

Let file `sum_vec.mpc` contain the following mpC code

```
#include <stdio.h>
nettype Star(n) {coord I=n;};
#define M 4  /*number of the processors*/
#define N 3 /*dimension of the vectors on the each processor*/
#define NM N*M   /*dimension of the source vectors*/
void [*]main() {
  int [host]x[NM], [host]y[NM], [host]z[NM], [host]i;
  void [*]parsum(),[*]parsum1();
  ([host]printf)("<host input vectors>\n");
  for(i=0;i<NM;i++) {
    x[i]=i;
    y[i]=-i;
  }
  ([host]printf)("x=");
  for(i=0;i<NM;i++)([host]printf)(" %d",x[i]);
  ([host]printf)("\ny=");
  for(i=0;i<NM;i++)([host]printf)(" %d",y[i]);
  ([host]printf)("\n");
  parsum((void*)x, (void*)y, (void*)z);
  ([host]printf)("<host result vector>\n z=");
  for(i=0;i<NM;i++)([host]printf)(" %d",z[i]);
  ([host]printf)("\n");
}
void [*]parsum(int [host]x[M][N], int [host]y[M][N],
               int [host]z[M][N])
{
  net Star(M) Sn;
  int [Sn]dx[N], [Sn]dy[N], [Sn]dz[N];
  int [Sn] i,[host]j,[host]l;
  dx[]=x[];
  dy[]=y[];
  dz[]=dx[]+dy[];
  z[]=dz[];
}
```

The program sums up two vectors on `M` virtual processors using function `parsum`. The function creates a network with `M` virtual processors, scatters portions of source vectors, calculates sum, and gathers results to the host. In addition, let us suppose that the user works on a workstation named `beta`. Let the user want to execute the application on this single workstation as DMM, and the corresponding VPM has not been created yet. Therefore the user should create the VPM. To do it, he creates description file `beta5` in $MPCTOPO directory containing the following 2 lines:

```
# my own workstation only
beta 5
```

Then the user creates the VPM with name `beta5` by typing:

```
mpccreate beta5
```

On the console of `beta` something like

```
mpccreate: net definition /home/mpc/topo/beta5.def is created.
mpccreate: scheme /home/mpc/topo/beta5.ts is created.

LAM 6.0 - Ohio Supercomputer Center

mpccreate: wait for creation 'beta5'
mpccreate: parallel machine 'beta5' is created.
```

will appear. Note that immediately after creation, VPM is opened. Then the user compiles his mpC file:

```
mpcc sum_vec.mpc
```

and copies output file `sum_vec.c` to the $MPCLOAD directory:

```
cp sum_vec.c $MPCLOAD
```

Because the user wants to execute the application on his workstation only, he can pass the step of broadcasting `sum_vec.c`. Then he makes executable `sum_vec`:

```
mpcload -o sum_vec sum_vec.c
```

Finally, the user runs the application:

```
mpcrun sum_vec
```

On the console of `beta` something like:

```
<host input vectors>
x= 0 1 2 3 4 5 6 7 8 9 10 11
y= 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11
<host result vector>
```

```
 z= 0 0 0 0 0 0 0 0 0 0 0 0
```

will appear. If user wants to check status of `beta5,` he may type

```
mpctouch
```

On the console of `beta` something like

```
mpctouch:   Ready!
```

will appear.

## 10.2 Complicated example.

Let the virtual parallel machine to run the application `gal-buf.mpc` has been already opened. To produce two target C files - the first for the virtual host processor executing code that includes calls to Xlib displaying data in the graphical form, and the second for the rest of virtual processors not involved in graphical representing data, one can type:

```
mpcc -I/usr/openwin/include -het gal-buf.mpc
```

Note. Use the absolute application name if `gal-buf.mpc` is not in the current directory. Use directory other then `/usr/openwin/include` if necessary (that is, use the directory where X Windows system holds its include files on the host workstation).

The above command will produce files `gal-buf_host.c` and gal-buf_node.c in the current directory. To make these files accessible to the mpC programming environment, one should copy them into the $MPCLOAD directory:

```
cp gal-buf_host.c gal-buf_node.c $MPCLOAD
```

To broadcast these files from the host workstation to all workstations constituting the distributed memory machine, one should type:

```
mpcbcast gal-buf_host.c gal-buf_node.c
```

To produce executable `gal-buf` on each workstation of the distributed memory machine, one can type:

```
mpcload -het -o gal-buf gal-buf.c -lm -host \
-L/usr/openwin/lib -lX
```

Note. Use a directory other then `/usr/openwin/lib` if necessary (that is, use the directory where X Windows system holds its libraries on the host workstation). Use an option other then `-lX` if necessary (that is, use the proper name for the X library; it may be `-lX11` or something else).

Finally, to run the application, one can type:

```
mpcrun gal-buf -- input_file
```

where file `input_file` contains input data for the application.

**Note**. Use the absolute name of the input file if it is placed in a directory other then the directory which was a current directory when you open your virtual parallel machine.