PARADIX

User Guide

Table of context

1. Welcome to ParaDiOx

ParaDiOx is an application built to help mathematicians in their daily work of solving big problems using Ox. By collaborating with other mathematicians and use their resources, e.g. their computer power, this application makes calculations a lot faster by taking advantage of parallel functions.

1.1. What's the purpose of this user guide?

This User Guide is written for you to get a good start with ParaDiOx at an early stage and also enlighten you of all the possibilities and opportunities that come with ParaDiOx. When you have read this guide you should be able to install, run and understand how to make your own OX files that support MPI!

1.2. About Us

ParaDiOx project has been created by five students at Royal Institute of Technology, Stockholm Sweden (also known as KTH). These students are Andreas Öhrvall, Dennis Frantzén, Martin Edquist, Michael Bergman and Peter Stolt. We all knew each other from earlier courses and projects. Hence, we have learned to work together in an efficient way. We found this to be very useful in this project.

2. Installing ParaDiOx

2.1. Before you start

The installation of ParaDiOx is separated in two different installation types - the slave installation and the master installation. When setting up a slave computer, you should use a slave installation and subsequently when setting up a master computer you should always use the master installation. The slave installation is also separated into two different types of automation. So the main option when installing ParaDiOx is to choose between the following files:

1. *MasterSetupAuto.exe*. This file will install all components necessary to run a computer as a master. The installation program will provide help to create an account, installing an mpich daemon etcetera. However, the installation program needs to install VB-runtime dlls, if these dll files aren't already installed on the computer.

2. *MasterSetupClean.exe*. The difference between this installation and the previous master installation is that this installation doesn't require VB-runtime dlls. On the other hand, you have to create the account manually in this installation. The main drawback with the requirement of installing VB-runtime dlls is that you will need to restart the computer before you can continue with the installation. But if you already have VB-runtime dlls installed its only benefits with using the previous slave installation instead of this one.

3. *SlaveSetupAuto.exe*. This file will only install components necessary to run a computer as a slave. Similar to the MasterSetup, it will provide automatic account creation etcetera and it also requires VB-runtime dlls.

4. *SlaveSetupClean.exe*. The difference between this installation and the previous slave installation is that this installation doesn't require VB-runtime dlls. On the other hand, you have to create the account manually in this installation. The main drawback with the requirement of installing VB-runtime dlls is that you will need to restart the computer before you can continue with the installation. But if you already have VB-runtime dlls installed its only benefits with using the previous slave installation instead of this one.

2.1.1. System requirements

- IBM-compatible computer running Windows NT 4.0, Windows 2000 or Windows XP
- The programming environment Ox.

2.2. Installation instructions

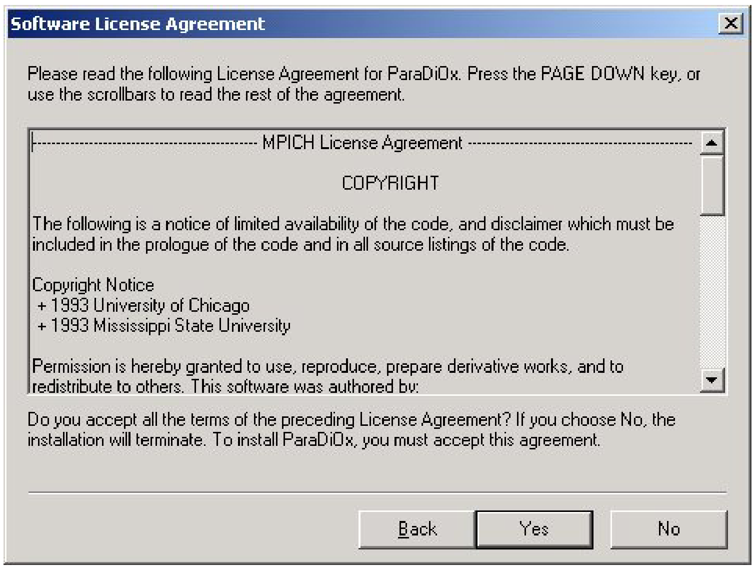
Once you have chosen desired installation file, it is time to start installing ParaDiOx. Since the installation instructions for the different installation files are quite similar, they will be presented as one instruction and where the installation files differ it will be shown as *MasterAuto*, *MasterClean*, *SlaveAuto* or *SlaveClean*.

1. Unzip the files into a temporary directory. Choose Run from the Start menu. The Run dialog box will appear.

2. Type <Path to the temporary directory> \MasterInstAuto\setup.exe in the text box provided and then click OK. Here, MasterInstAuto is the path to the *MasterAuto* installation. If it's the *SlaveAuto* installation you are installing this path is replaced with SlaveInstAuto. Similar for the *MasterClean* or the *SlaveClean* installation the path is replaced with MasterInstClean respectively SlaveInstClean. For example if you have chosen the temporary directory to C:\ParaDiOx\Installation and is installing *SlaveClean* installation the path would be C:\ParaDiOx\Installation\SlaveInstClean\setup.exe.

3. (*MasterAuto*, *SlaveAuto*) The installation program will start with a pre-installation of VB-runtime dlls. It will check if all necessary files are installed; if not it will install these files. If any VB-runtime files are installed, the installation requires a restart of your computer and you will need to run the setup program again after the restart. Otherwise the installation program will continue to the next step.

   (*MasterClean*, *SlaveClean*) This installation doesn't require this step, so you can move forward to the next step.


Software License Agreement

Please read the following License Agreement for ParaDiOx. Press the PAGE DOWN key, or use the scrollbars to read the rest of the agreement.

-- MPICH License Agreement --

COPYRIGHT

The following is a notice of limited availability of the code, and disclaimer which must be included in the prologue of the code and in all source listings of the code.

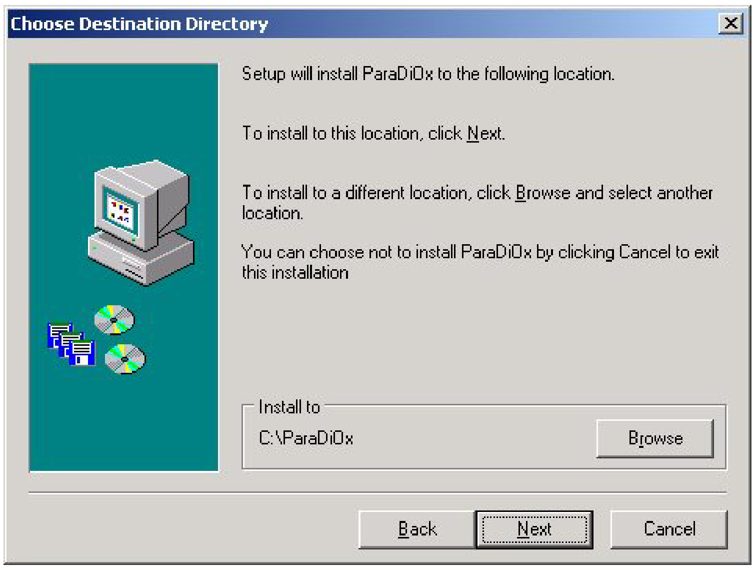
Copyright Notice
+ 1993 University of Chicago
+ 1993 Mississippi State University

Permission is hereby granted to use, reproduce, prepare derivative works, and to redistribute to others. This software was authored by:

Do you accept all the terms of the preceding License Agreement? If you choose No, the installation will terminate. To install ParaDiOx, you must accept this agreement.

Figure 2-1. The license agreement

4. (*All setups*) You will now get some information about the installation and on the next screen you will find license agreement on the product MPICH since ParaDiOx is a further development of MPICH. To continue the installation you will need to select "yes" to accept on this agreement. By selecting "No" you will exit the installation.


Choose Destination Directory

Setup will install ParaDiOx to the following location.

To install to this location, click Next.

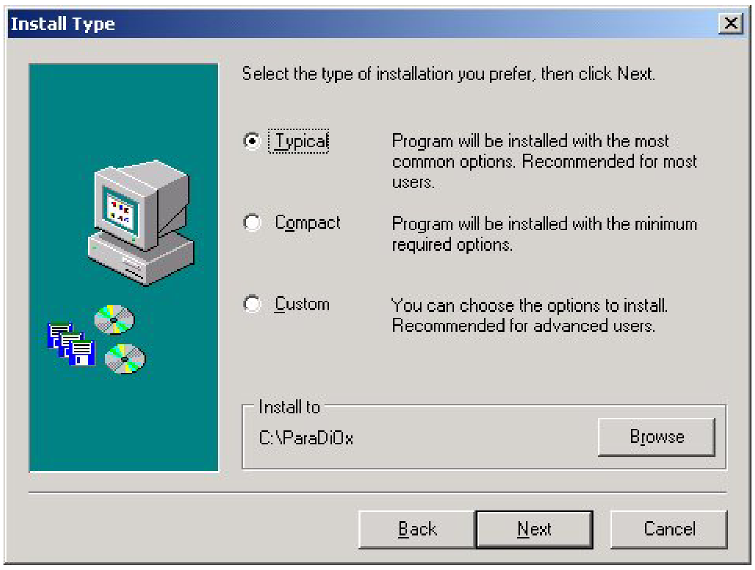
To install to a different location, click Browse and select another location.

You can choose not to install ParaDiOx by clicking Cancel to exit this installation

Install to
C:\ParaDiOx

Figure 2-2. Choosing destination directory

5. (*All setups*) Now you need to choose an installation directory. Recommended is that you choose the same directory on all computers you are going to use. It's possible to select different installation directories but you may come across difficulties in setting up your configuration on the master computer.


Install Type

Select the type of installation you prefer, then click Next.

[✓ Typical] Program will be installed with the most common options. Recommended for most users.

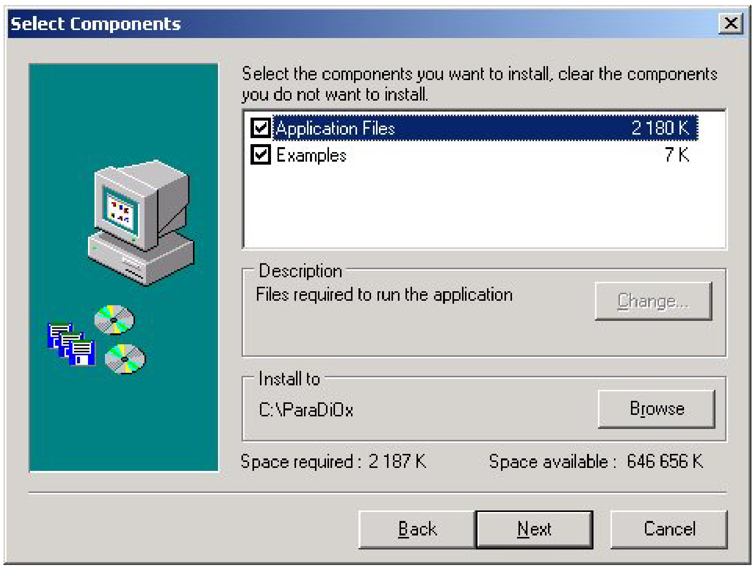
Compact Program will be installed with the minimum required options.

Custom You can choose the options to install. Recommended for advanced users.

Install to
C:\ParaDiOx

Figure 2-3. The installation type panel

6. (*All setups*) There are three different types of installations. The typical installation will install all the files that are needed as well as example files. The compact installation will only install the necessary files. If you choose the custom installation you will get to a new screen that lets you customize your installation with the package you want to install.


Select Components

Select the components you want to install, clear the components you do not want to install.

☑ Application Files	2 180 K
☑ Examples	7 K

Description
Files required to run the application

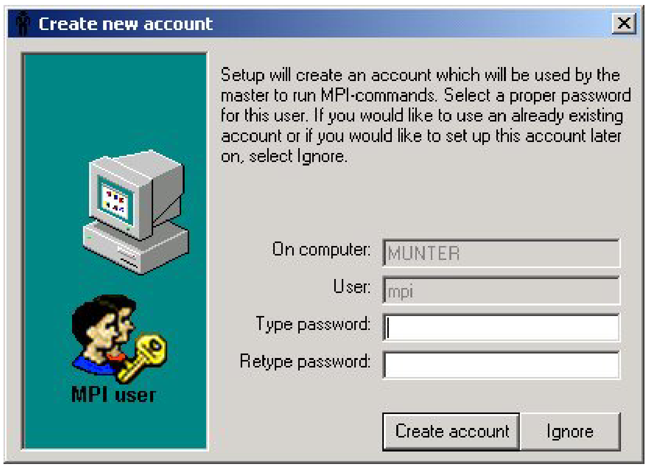
Install to
C:\ParaDiOx

Space required : 2 187 K   Space available : 646 656 K

Figure 2-4. Customizing your installation

7. (*All setups*) Next you will need to choose where the program links will be placed in the start menu and then you can start installing the files.

8. (*All setups*) After the files have been installed on your computer, a command prompt will appear that installs the MPICH daemon as a service.


Create new account

Setup will create an account which will be used by the master to run MPI-commands. Select a proper password for this user. If you would like to use an already existing account or if you would like to set up this account later on, select Ignore.

On computer: MUNTER
User: mpi
Type password:
Retype password:

Figure 2-5. Creating new user account

9. (*MasterAuto*, *SlaveAuto*) If you are running Windows 2000 or Windows XP you now need to setup an account for the master computer by choosing a proper

password. The account will be a member to the group “Guests”. If you are running Windows NT 4 you will need to setup this account manually in the end of this installation instruction.

(*MasterClean*, *SlaveClean*) Move to the next step.

10. (*All setups*) Restart your computer to get the service running.

11. (*MasterAto*, *SlaveAuto*) Now the installation is finished. You can now use ParaDiOx.

    (*MasterClean*, *SlaveClean*, *Windows NT 4*) Add a new user by using the User manager on the start menu and add the user to the group “Guests”. Then the installation is finished. You can now use ParaDiOx.

2.2.1. How to add a new user in Windows NT 4

1. Start the User Manager as shown in figure 2-6 below.

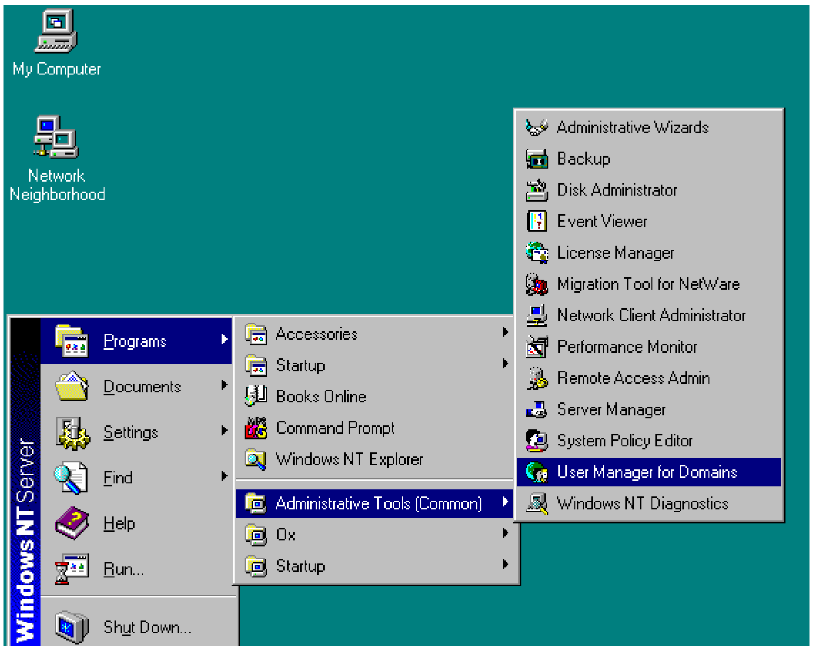
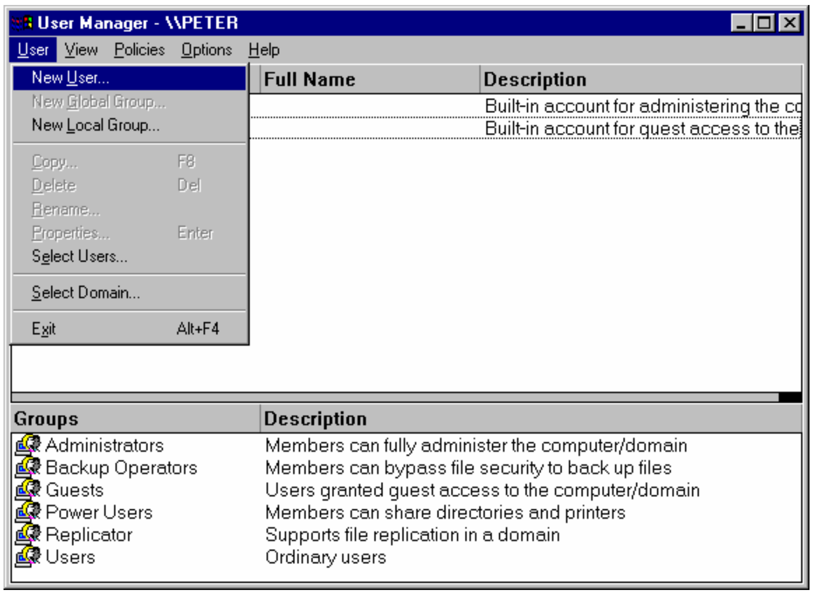


Figure 2-6. Starting the User Manager

2. Select new user from the user menu as depicted in figure 2-7.

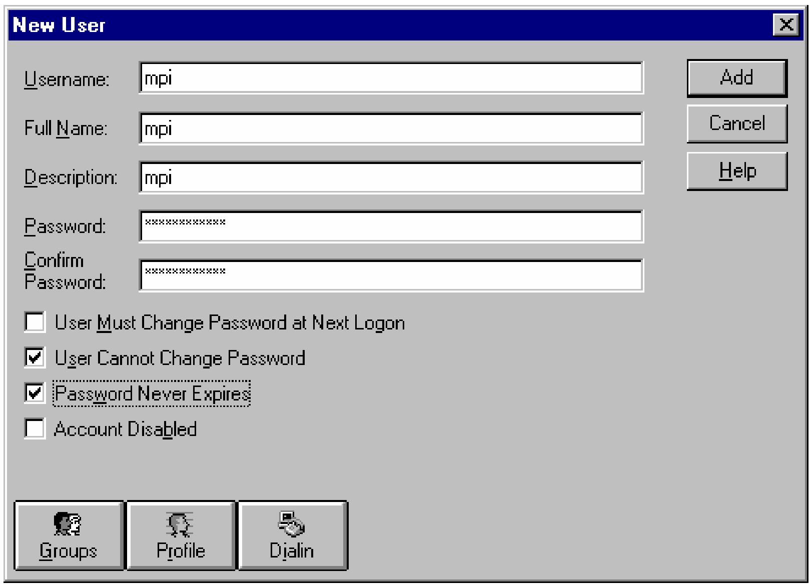


Full Name	Description
	Built-in account for administering the co
	Built-in account for guest access to the

Groups	Description
Administrators	Members can fully administer the computer/domain
Backup Operators	Members can bypass file security to back up files
Guests	Users granted guest access to the computer/domain
Power Users	Members can share directories and printers
Replicator	Supports file replication in a domain
Users	Ordinary users

Figure 2-7. Selecting New User from the User menu

3. Choose desired selections as illustrated in figure 2-8.



Username: mpi
Full Name: mpi
Description: mpi
Password: *************
Confirm Password: *************

☐ User Must Change Password at Next Logon
☑ User Cannot Change Password
☑ Password Never Expires
☐ Account Disabled

Figure 2-8. Creating new user account

4. Click on the Groups button in previous figure and make the user a member of Guests as described in figure 2-9.

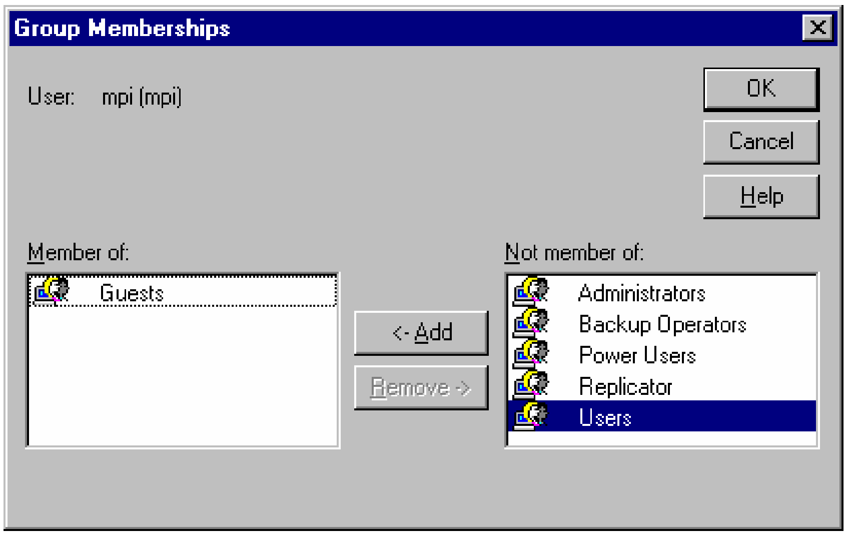
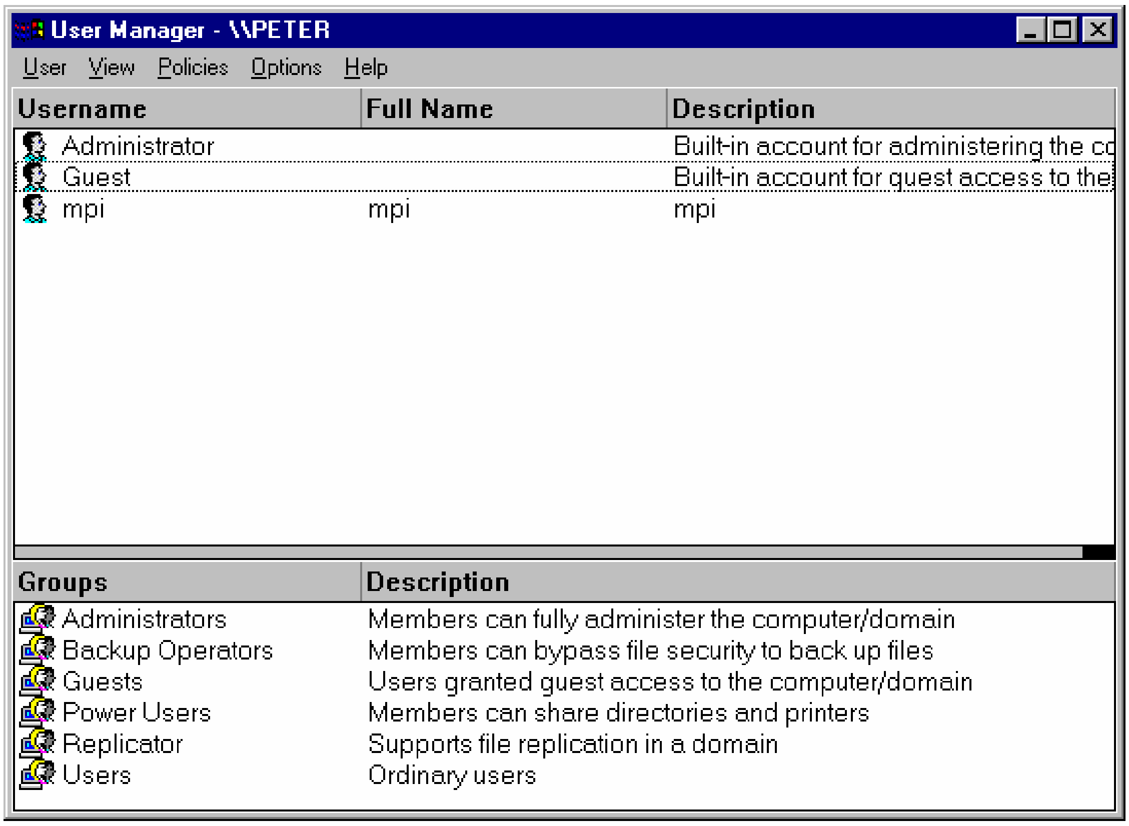


Figure 2-9. Mpi is member of the Guests group

5. Then you get back to figure 2-9. Here you click on the Add button. The result should be similar to figure 2-10!



User Manager - \\PETER

Username	Full Name	Description
Administrator		Built-in account for administering the co
Guest		Built-in account for guest access to the
mpi	mpi	mpi

Groups	Description
Administrators	Members can fully administer the computer/domain
Backup Operators	Members can bypass file security to back up files
Guests	Users granted guest access to the computer/domain
Power Users	Members can share directories and printers
Replicator	Supports file replication in a domain
Users	Ordinary users

Figure 2-10. The result when the mpi user is added.

2.2.2. How to add a new user in Windows 2000

1. Open the Control Panel item by clicking on the Start menu, point to Settings and click on Control Panel.

2. Open Users and Passwords in Control Panel.

3. Click on the button Add.

4. Select a proper username, for example mpi (selecting a real name and a description is optional) and then click next.

5. Select a password (to make it easier to remember the password it is recommended to have the same password on all the slaves) and then click next.

6. Make the user a member of the group Guests by selecting the last of the optional buttons and then scroll in the list of groups until you find Guests. Now you only need to confirm and then you can start using the account.

   Note: If you having trouble with adding a new user, please read the Windows 2000 manual.

3. Learning how to use ParaDiOx

3.1. Preface

It is strongly suggested to get the example program working before starting to develop your own Ox program. When this part is working, the next step is to develop your own Ox code. You can execute several processes on one workstation and test so everything works properly before executing on several computers. When your program is developed, you need to spread the file to all installed computers or execute it over the network over a Windows share. It is possible to use the mpd (mpich) to spread the file. For more info about that, please read the mpd manual.

The graphical user interface is an improved version of the mpich graphical user interface. Since ParaDiOx is fully compatible with everything made for mpich it is possible to use the original graphical user interface or MPIrun. You can find everything about those programs in the MPICH manual. You can also find information about how to write your own configuration file here. This could be useful if you want to put the files on different locations on different computers for example.

Our graphical user interface is also compatible with most mpich applications.

3.2. Getting started

This chapter is a guide for those who neither have used ParaDiOx, nor is a regular user of Ox to. The purpose is to explain what it really is, in order to help the user to get the system up and running. It will provide information of the link between ParaDiOx and Ox, so if you are looking for a description how to make general Ox programs, we refer to the Ox manual.

The first thing you will have to do is to get your hands on a couple of computers. It is possible to run ParaDiOx on only the local computer, but since this program is about enhancing performance via distributed calculation, the point with a local run seems less clear. However, the computers you are going to use all need to get an installation of ParaDiOx and also the Ox environment. The Ox environment is freely available on the homepage of Ox (http://www.nuff.ox.ac.uk/Users/Doornik/index.html) for academic research; others may need to pay a fee. Please do regular installations of the Ox Console on all the computers. If you are unsure of the procedure on installing the Ox environment, you better look in the Ox manual. A comment to the Ox installation is that there will be a lot easier if you choose the same path on all installations, but it is possible to have different paths. When the Ox environment is installed on all computers it is time to move forward to the ParaDiOx installation. There are two main differences between the installation files for ParaDiOx: the Master installation and the Slave installation. The computer you will use as you working environment, i.e. the computer where you will distribute your calculations from, will need to have a Master installation. On the other hand, all the rest of the computers which only will perform calculations and return the corresponding results, will need to have a Slave installation. Please read the *Installing ParaDiOx* chapter for further instructions on that area.

Now when the installation is finished it is time to get on with the Ox programming. In this manual we will only cover a small example of Ox code because a full description of the Ox environment would require an entire manual. So if want further information about programming in Ox we recommend you to read the Ox manual.

3.2.1. Programming in Ox

The problem we are going to exemplify is the following (figure 3-1): A master computer is going to calculate an equation, using a parallel algorithm. It sends a part of the calculation (the red message) to the first slave and when the slave has finished his calculation, the master will receive the answer from it. Similar messages are sent to the other slave computers, which in turn will process their calculations at the same time. The answers are then post processed by the master. This will enhance the calculation time (but in this trivial case, the network communication will eat up all time we have won in enhanced calculation performance).

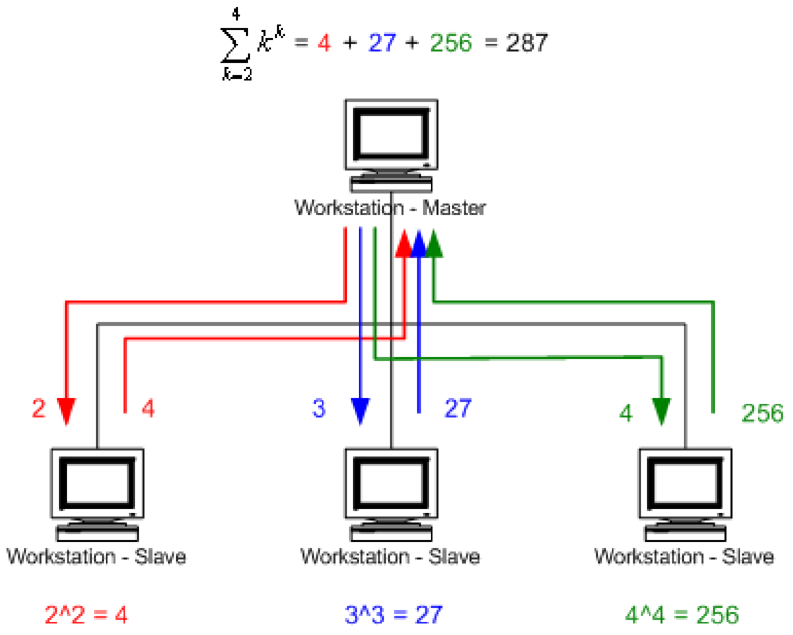


Figure 3-1 A trivial calculation example

We now need to create an .ox file. The simplest way if doing this is to open Notepad, select “Save as…” in the menu and then select a directory where the file should be placed, e.g. “c:\temp\”. Then in the file type scrollbar, select all files. Then choose a name for the file you want to create followed by “.ox”; in this case we will call it “helloworld.ox”. Finally select Save.

Now we can start programming our example. You can also do this in Notepad, but of course you may use any text editor.

In the beginning of the ox-file you specify files that should be included in the Ox program, pretty much like C or C++ (or even Java) if you have experience from those programming language. This specification is typed as following:

```
#include <oxstd.h>
```

In this case the included file is a standard Ox library with methods that may be used in the programming code. Next we will declare some methods that are used for the communication with ParaDiOx (MPICH).

```
extern "ox2mpich,Init" MPI_Init();
extern "ox2mpich,Comm_size" MPI_Comm_size();
extern "ox2mpich,Comm_rank" MPI_Comm_rank();
extern "ox2mpich,Get_processor_name" MPI_Get_processor_name( tmp);
extern "ox2mpich,Bcast" MPI_Bcast(const message,const root);
extern "ox2mpich,Reduce" MPI_Reduce(const SENDBUF,const RECVBUF,const
OP,const ROOT);
extern "ox2mpich,Finalize" MPI_Finalize();
extern "ox2mpich,Wtime" MPI_Wtime();
extern "ox2mpich,Send" MPI_Send(const SENDBUF, const target, const tag);
extern "ox2mpich,Recv" MPI_Recv(const SENDBUF, const source, const tag);
extern "ox2mpich,Probe" MPI_Probe(const source, const tag);
extern "ox2mpich,Iprobe" MPI_Iprobe(const source, const tag, const
flag);
```

All these methods that we have declared here may now be used in the code. These methods are specified in detail later on, in chapter 5. Continuing with the programming, we need to start thinking of the logic of the program. As all runable Ox program, they all need a main procedure, which will be executed.

```
main() {
    // program logic here
}
```

These simple lines of code will do it and we can start coding between the brackets. The first order of business is to initialize and finalize the MPI connection. All Ox programs which are going to communicate via ParaDiOx will need this kind of "start and end declaration".

```
main() {
    MPI_Init();
    // program logic here including mpi messaging
    MPI_Finalize();
}
```

In Ox you declare variables you are going to use by using the code “decl” followed by the variable name. There is no difference between different types of variables, so both strings and integers etcetera are declared in the very same way. We will at least need to have two variables declared. Those are myId and numProcs. The variable myId will contain a unique integer which identifies the process while the variable numProcs contain the number of available processes (both are defined during the magical initialization procedure). We recommend to set these values at the same time as the declaration by calling the MPI methods at the very same line of code, as shown in the next code example.

```
main() {
    MPI_Init();
    decl numProcs = MPI_Comm_size();
    decl myId = MPI_Comm_rank();
    // program logic here including mpi messaging
    MPI_Finalize();
}
```

A number of processes are started on a number of computers, and the only thing which is interesting when you programming the Ox programs are how many processes you are able to distribute to. Therefore we will need numProcs. Each of those processes has a unique number, and for the current process it is myId. We will now need to decide which process will be the master process. Don’t be confused with the definitions of master computer and master process, because there is a difference between them. The master computer is where you will run the graphical user interface and where you will decide when to start the ox program etcetera. Meanwhile the master process is the process which will act as a message manager to send tasks to slave processes and to do the post calculations. Although in most cases, the master process and the master computer is the same. It is somewhat of a standard to choose the master process to be process zero.

```
main() {
    MPI_Init();
    decl numProcs = MPI_Comm_size();
    decl myId = MPI_Comm_rank();
    if (myId == 0) {
        // master program logic
    } else {
        // slave program logic
    }
    MPI_Finalize();
}
```

We have now separated the program logic in slave logic and master logic. Now we can start concentrating on the algorithm design. So let’s start with the slave algorithm. The slave processes should be designed so that they receive an integer x from the master process, calculate the integer x with the formula x^x and then return the answer to the master process. This can be done with the following code:

```
decl i, message = 0, resultCalc = 1;
MPI_Recv(&message, 0, 1);
for (i = 1; i <= message; i++) {
    resultCalc = resultCalc * message;
}
print("\nSlave result: ", message, "^", message, "=", resultCalc);
MPI_Send(&resultCalc, 0, 1);
```

We use MPI_Recv to receive the integer from the master, and use MPI_Send to return the answer to the master. The algorithm uses a for-loop, which notation is very similar to other programming language, to perform an exponential calculation. We also use a print line so we know the result of the process. An interesting note is that all print-statements only will be exposed on the master computer. So now we need to create the logic for the master process.

The master process should distribute all calculations to the slave processes and then wait for answer from them. A possible way of doing this is to loop through all processes and send integers to them and order them to work with it and then start a loop that waits until we have received answer from all the slave processes. So let’s do so.

```
decl j, result=0, message=0, answers=0;
decl answerFromSlave = zeros(1, numProcs);
for (j = 2; j < numProcs + 1; j++) {
    MPI_Send(&j, j - 1, 1);
}
while (answers < numProcs - 1) {
    for (j = 1; j < numProcs; j++) {
        MPI_Iprobe(j, 1, &result);
        if (result != 0) {
            MPI_Recv(&message, j, 1);
            answerFromSlave[j] = message;
            answers++;
        }
        result = 0;
    }
}
```

What do we have left? Well, we must represent our answers from the hosts in some way and we also need to post calculate the answers. Hence, we add this code at the end of the master's logic.

```
print("\nMaster result: ",answerFromSlave[1]);
result = answerFromSlave[1];
for (j = 2; j < numProcs; j++) {
    print("+", answerFromSlave[j]);
    result = result + answerFromSlave[j];
}
print("=", result, "\n");
```

Now we soon are finished - just a small detail left. Since the code we have produced requires at least two processes, one master and one slave process, we need to handle the case when only one process is selected. A simple way to come around this problem is to put an if-statement to check this condition.

```
if (numProcs == 1) {
    print("\nNumber of processes should be more then 1,");
    print("\nplease increase number of processes!");
} else {
    // Master and slave logic here
}
```

Now let's make it a complete program and se what we have accomplished:

```
#include <oxstd.h>
extern "ox2mpich,Init" MPI_Init();
extern "ox2mpich,Comm_size" MPI_Comm_size();
extern "ox2mpich,Comm_rank" MPI_Comm_rank();
extern "ox2mpich,Get_processor_name" MPI_Get_processor_name( tmp);
extern "ox2mpich,Bcast" MPI_Bcast(const message,const root);
extern "ox2mpich,Reduce" MPI_Reduce(const SENDBUF,const RECVBUF,const OP,const ROOT);
extern "ox2mpich,Finalize" MPI_Finalize();
extern "ox2mpich,Wtime" MPI_Wtime();
extern "ox2mpich,Send" MPI_Send(const SENDBUF, const target, const tag);
extern "ox2mpich,Recv" MPI_Recv(const SENDBUF, const source, const tag);
extern "ox2mpich,Probe" MPI_Probe(const source, const tag);
extern "ox2mpich,Iprobe" MPI_Iprobe(const source, const tag, const flag);

main() {
    MPI_Init();  // Initialize MPI
    decl numProcs = MPI_Comm_size();
    decl myId = MPI_Comm_rank();
    if (numProcs == 1) {
        print("\nNumber of processes should be more then 1,");
        print(“\nplease increase number of processes!");
    } else {
        if (myId == 0) {
            decl j, result=0, message=0, answers=0;
            decl answerFromSlave = zeros(1, numProcs);
            for (j = 2; j < numProcs + 1; j++) {
                MPI_Send(&j, j - 1, 1);
            }
            while (answers < numProcs - 1) {
                for (j = 1; j < numProcs; j++) {
                    MPI_Iprobe(j, 1, &result);
                    if (result != 0) {
                        MPI_Recv(&message, j, 1);
                        answerFromSlave[j] = message;
                        answers++;
                    }
                    result = 0;
                }
            }
            print("\nMaster result: ",answerFromSlave[1]);
            result = answerFromSlave[1];
            for (j = 2; j < numProcs; j++) {
                print("+", answerFromSlave[j]);
                result = result + answerFromSlave[j];
            }
            print("=", result, "\n");
        } else {
            decl i, message = 0, resultCalc = 1;
            MPI_Recv(&message, 0, 1);
            for (i = 1; i <= message; i++) {
                resultCalc = resultCalc * message;
            }
            print("\nSlave result: ", message, "^", message, "=", resultCalc);
            MPI_Send(&resultCalc, 0, 1);
        }
    }
    MPI_Finalize();  // Finalize MPI
}
```

Now we have a program which will calculate the formula up to the number of processes we choose in the graphical user interface. Next thing to do is to try it in ParaDiOx.

3.2.2. Getting helloworld.ox to run

This section will concentrate on getting the program we created in the previous section running on the computer network we installed in the beginning of this chapter.

First you need to copy the *helloworld.ox* to all computers in your network. It is preferable that the files get the same path. In this example we will assume that all paths are the same. Otherwise you need to create a configuration file. When you have distributed the ox-file to all computers you need to get knowledge of the computer name on the slaves. Please view the Windows manual if you don't know how to get it. Next it is time to start running the helloworld example.

Begin with starting up ParaDiOx Graphical User Interface (GUI) on the master computer via the start menu or by executing *ParaDiOx.exe*, and the GUI will appear. Now you need to write in the path to your oxl.exe file followed by a space and then the path to your *helloworld.ox* in the *Application* input field. In our case this would be:

```
c:\Ox\bin\oxl.exe c:\temp\helloworld.ox
```

Next you will need to setup available computers. Mark the radiobutton named *Hosts* and the input field below will be enabled. Write in the field that became enabled the computer name of the first slave computer. Then press on the *Add* button and the computer name will be added to the list below. When you have added the computers, you will need to select which computers you want to run on by marking them with a mouse click right on the computer name in the list.

Now it is only thing left to do is to choose how many processes you want to have. This can be accomplished in the *Number of processes* input field. And now you may press on the *Run* button. A new input box will appear requesting for Account and Password. If you have followed the installation notes you will input the account "mpi" and the password you selected during the installation and then press the *OK* button. The calculation will now start and depending on the number of chosen processes you will get an output similar to:

```
Ox version 3.10 (Windows) (C) J.A. Doornik, 1994-2002
This version may be used for academic research and teaching only

Ox version 3.10 (Windows) (C) J.A. Doornik, 1994-2002
This version may be used for academic research and teaching only

Ox version 3.10 (Windows) (C) J.A. Doornik, 1994-2002
This version may be used for academic research and teaching only

Ox version 3.10 (Windows) (C) J.A. Doornik, 1994-2002
This version may be used for academic research and teaching only

Slave result: 2^2=4
Slave result: 3^3=27
Slave result: 4^4=256
Master result: 4+27+256=287
```

You have now completed your first Ox program with ParaDiOx support. This helloworld example is very trivial, and do not provide false tolerant code etcetera. So now it is time to move forward to more advanced features. In the following sections of this chapter we will explain the GUI and give a more advanced example to run, called example.ox.

3.3. Graphical User Interface

In this section we will explain the different features in ParaDiOx Graphical User Interface. The thought is that you have learned the basic steps to build an Ox program by reading the earlier chapter, and now is ready to explore the strength of ParaDiOx with only some help on what the different buttons mean.

3.3.1. The main window

This section will explain the different parts of the Graphical User Interface main window. It will not provide specific details on how to use the main window, cause with little common sense you will get the feeling for it.

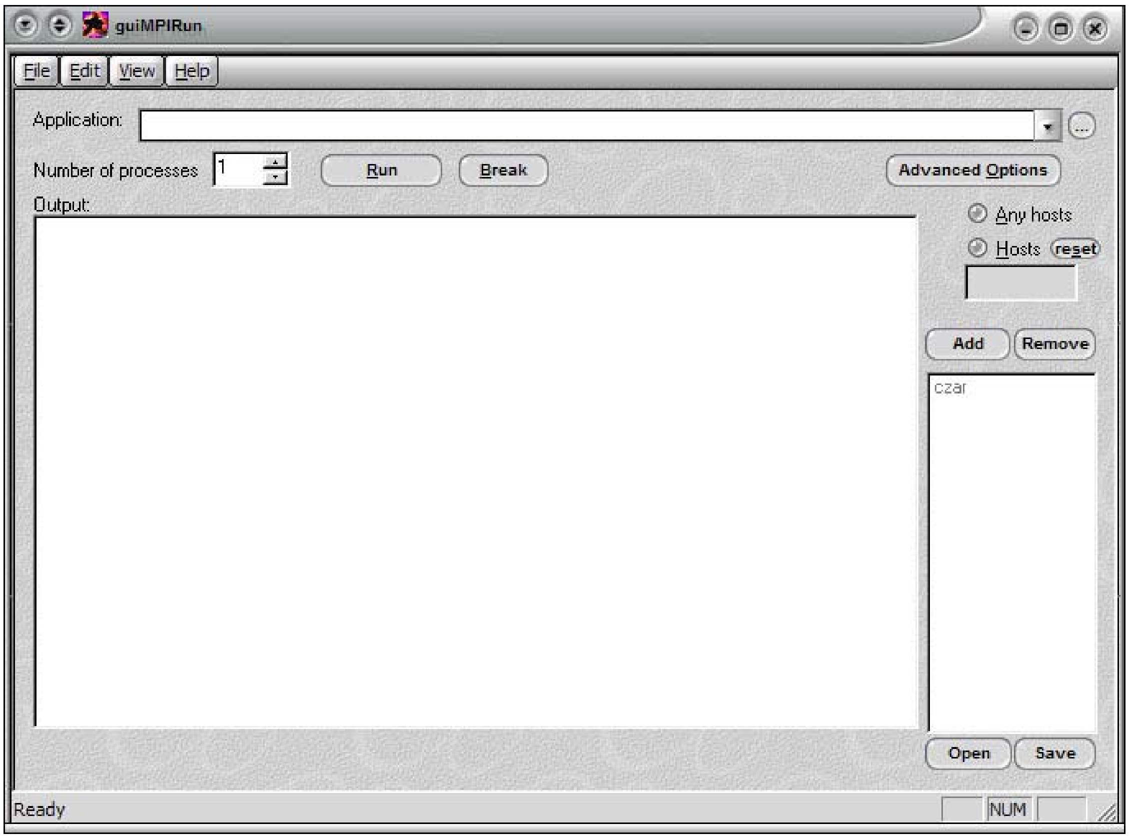


Figure 3-1. The main graphical user interface

Application

Enter the full path to the mpi application with any arguments. This can be a local or shared location. The path must be valid on all the nodes.
eg "c:\temp\myapp.exe arg1 arg2" or "\\myserver\myshare\myapp.exe arg1 arg2".

Number of processes

Select the number of processes you want to launch.

Run

Launch the mpich application.

Break

Kill the running application.

Any hosts

Run will choose from any of the hosts in the host list.

Hosts

Run only on the highlighted hosts from the list.

Reset

Reset the list of hosts to the list selected by MPIConfig.

Add button

Add a host to the list.

Remove button

Remove a host from the list.

Open button

Open a host list.

Save button

Save a host list.

Output

The output of the application shows up here. Ctrl+C will copy the output. You can also enter input here that will be sent to the root process.

3.3.2. Advanced options window

The advanced options window is providing options that will affect the running of a specific program. For example, if you have the Ox environment located differently on computers that you are going to use at the same run, you must support the program with a configuration file that explains where the oxl.exe file is located on each computer, likewise with the *.ox files.

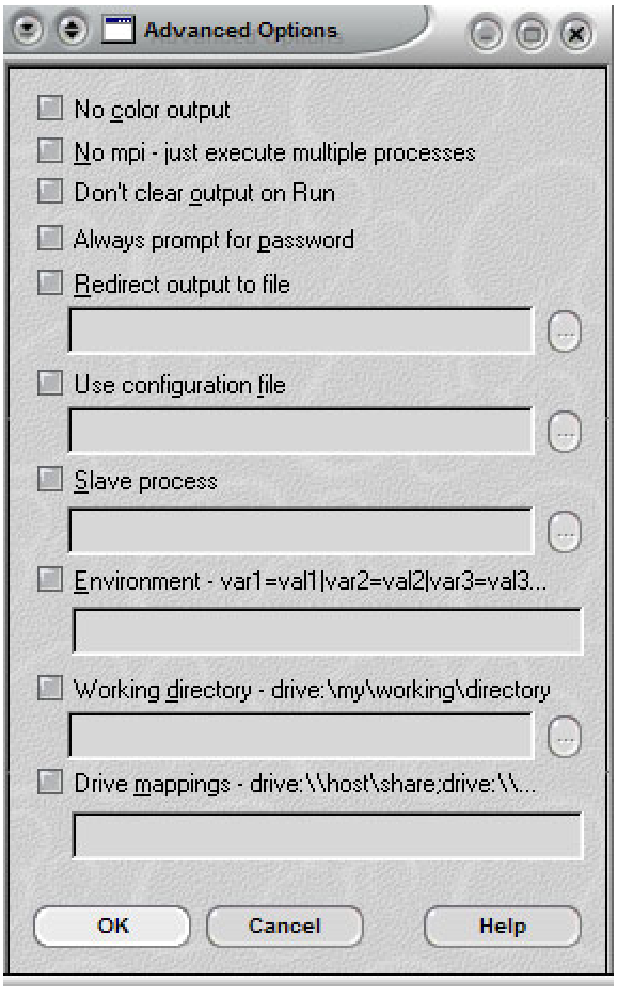

Advanced Options
No color output
No mpi - just execute multiple processes
Don't clear output on Run
Always prompt for password
Redirect output to file
Use configuration file
Slave process
Environment - var1=val1|var2=val2|var3=val3...
Working directory - drive:\my\working\directory
Drive mappings - drive:\\host\share;drive:\\...
OK
Cancel
Help

Figure 3-2. Advanced options window

No color output

The output will not be color coded according to the rank of the process

No mpi

Launch multiple processes that are not mpi applications - they never make any MPI calls.

Don't clear output on Run

The output in the output window does not get erased when the Run button is clicked.

Always prompt for password

Don't use the saved account in the registry, prompt for user and password every time 'Run' is selected.

Redirect output to file

Redirect the output of the mpi application to the specified file. The contents of the file will be deleted each time Run is selected.

Use configuration file

Select a configuration file to specify more complicated launching preferences.

Slave process

Specify the path to a second executable to be launched for every process except the root process. This is a quick way to launch a master/slave application without using a configuration file.

Environment: var1=val1|var2=val2|var3=val3|...varn=valn

This will set the environment variables specified in the string before each process is launched.

Working directory: drive:\some\path

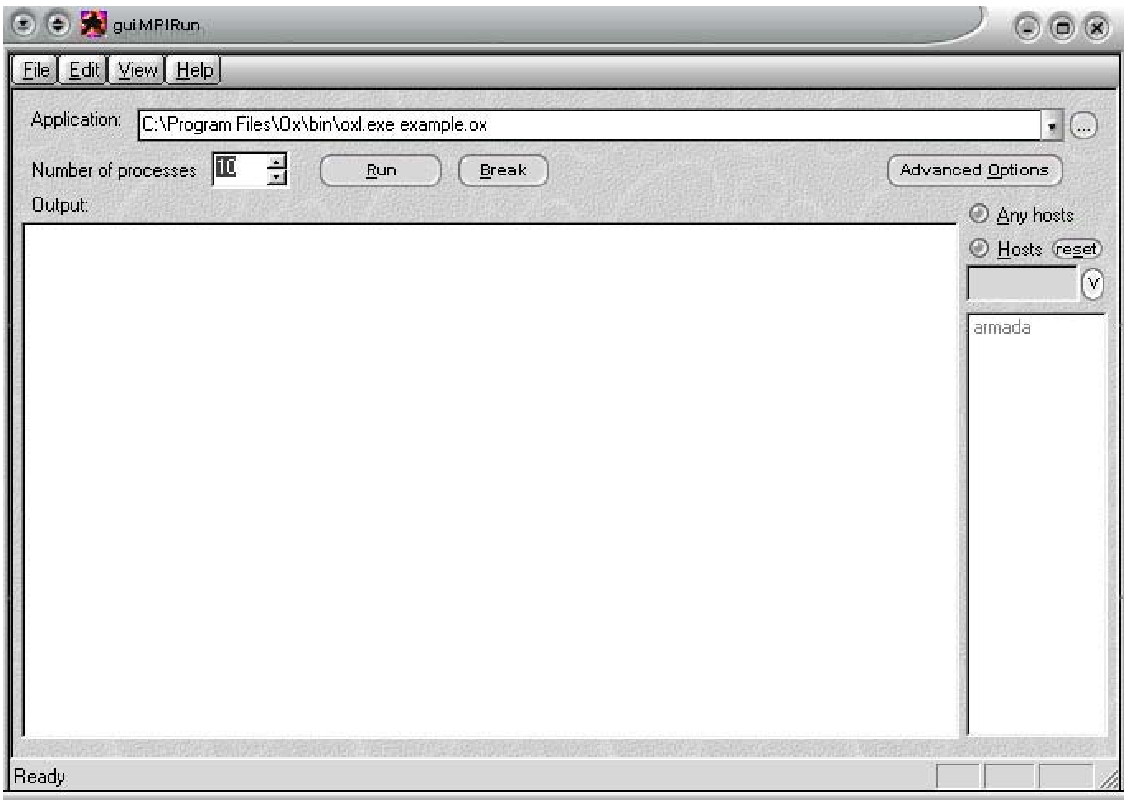
Set the working directory for the launched processes. If this option is not specified the current directory is used.

Drive mappings: drive:\\host\share

This option will map a drive on the hosts where the processes are launched. The mappings are removed after the processes exit. This option can be repeated multiple times separated by semi-colons. example: y:\\myserver\myapps;z:\\myserver\myhome

3.4. Run example

This section provides a simple step-by-step instruction how to start a distributed Ox program with Ox our DLL file and the MPICH system. (The thought is that you will use this section to get the more advanced example.ox running than the basic helloworld example).



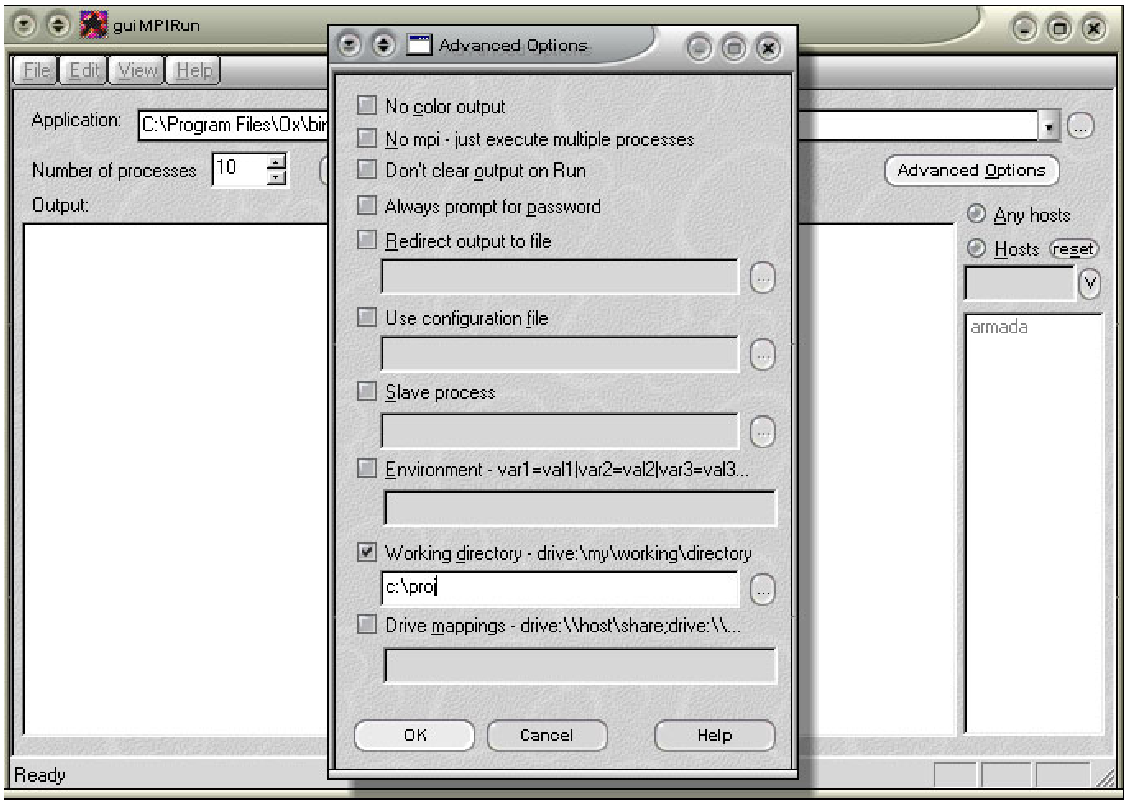
Application: C:\Program Files\Ox\bin\oxl.exe example.ox

Number of processes 10

armada

Figure 3-3. Enter path in the main graphical user interface

The first thing to do is to enter the path to your oxl executable with the complete path and the .ox program you want to run in the application field of the GUI.



Application: C:\Program Files\Ox\bir

Number of processes 10

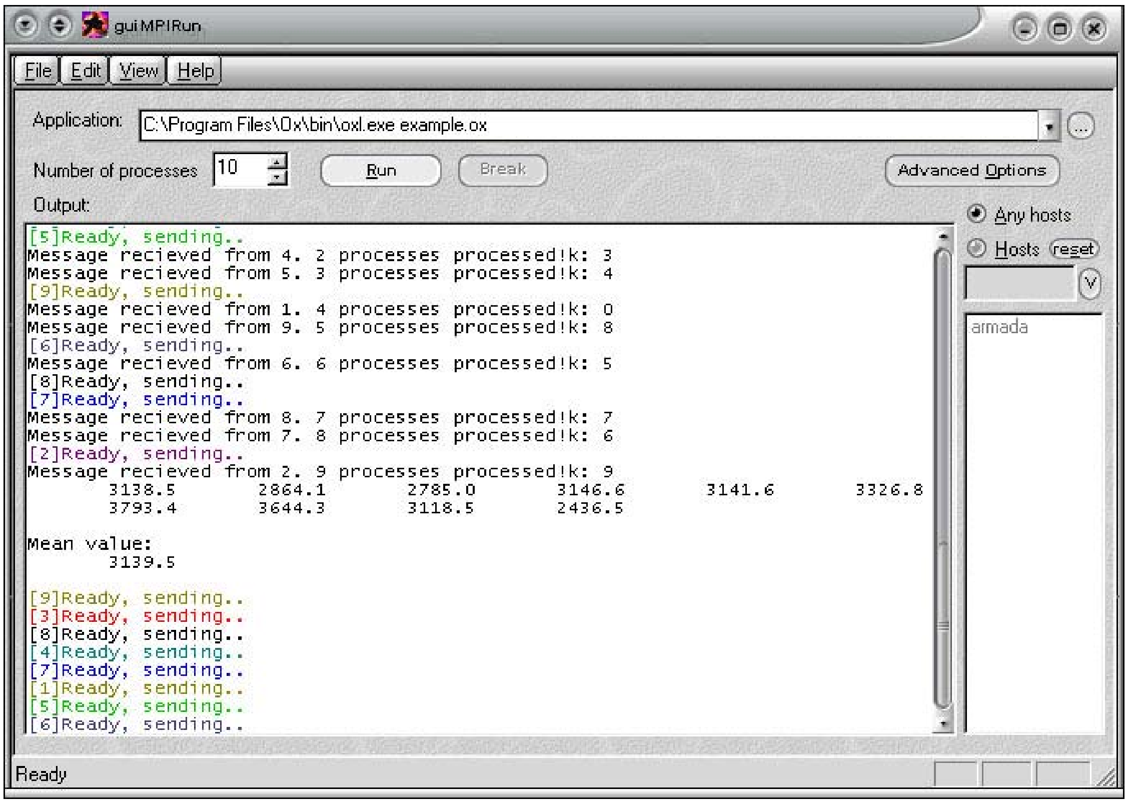
Output:

☐ No color output
☐ No mpi - just execute multiple processes
☐ Don't clear output on Run
☐ Always prompt for password
☐ Redirect output to file
☐ Use configuration file
☐ Slave process
☐ Environment - var1=val1|var2=val2|var3=val3...
☑ Working directory - drive:\my\working\directory
c:\pro
☐ Drive mappings - drive:\\host\share;drive:\\...

● Any hosts
● Hosts
armada

Figure 3-4. Enter path in advanced options window

The next step is to enter the advanced options and enter the path to your .ox files in the working directory field. Here you can also select the output to be redirected to a file instead of just being printed on the screen.



Application: C:\Program Files\Ox\bin\oxl.exe example.ox

Number of processes 10

Output:

```
[5]Ready, sending..
Message recieved from 4. 2 processes processed!k: 3
Message recieved from 5. 3 processes processed!k: 4
[9]Ready, sending..
Message recieved from 1. 4 processes processed!k: 0
Message recieved from 9. 5 processes processed!k: 8
[6]Ready, sending..
Message recieved from 6. 6 processes processed!k: 5
[8]Ready, sending..
[7]Ready, sending..
Message recieved from 8. 7 processes processed!k: 7
Message recieved from 7. 8 processes processed!k: 6
[2]Ready, sending..
Message recieved from 2. 9 processes processed!k: 9
       3138.5       2864.1       2785.0       3146.6       3141.6       3326.8
       3793.4       3644.3       3118.5       2436.5

Mean value:
       3139.5

[9]Ready, sending..
[3]Ready, sending..
[8]Ready, sending..
[4]Ready, sending..
[7]Ready, sending..
[1]Ready, sending..
[5]Ready, sending..
[6]Ready, sending..
```

Figure 3-5. Running processes

When all settings are done just select how many processes you want to run and on which computers you want the computation to run then press the “Run” button.

All output from all slave computers will then be redirected into the output field. By default they are color coded in order of process rank.

If the computation needs to be aborted just press the “Break” and all processes will die on the slave computers.

3.5. File copy guide by using the mpd console

This guide is a step by step instruction on how to distribute files to client computers with the MPICH system and the mpd console.

The two commands used are:

- `fileinit account=`*`x`*` password=`*`x`*

  Description:
  This command is the first command that must be issued before the other commands can be used. File operations are done under the security context of this user. If the password option is omitted, you will be prompted to input the password. Return values: nothing

- `putfile local=`*`fullfilename`*` remote=`*`fullfilename`*` replace=`*`yes/no`*` createdir=`*`yes/no`*

  Description:
  This command copies the file described by the local option to the location described by the remote option. Both the local and remote options must specify complete paths including file names. The replace and createdir options refer to the remote file. replace=*yes* overwrites the remote file if it exists. createdir=*yes* causes the path described by the remote option to be created if it doesn't exist. If replace and createdir are not specified, the defaults are replace=yes and createdir=yes. Return values: "SUCCESS" or "*error message*"

To start the mpd console launch mpd.exe with the –console switch and the name of the computer (in this example armada) the file should be copied to:

```
>mpd.exe –console armada
```

Upon successful connect issue the fileinit command with the username and password used during the installation of the Paradiox system, for example:

```
fileinit account=mpi password=mpich123
```

Then issue the putfile command with parameters:

```
putfile local=c:\temp\example.ox remote=d:\temp\example.ox replace=yes
createdir=yes
```

Then the file should be copied onto the remote computer and be ready to run thru guiMPIrun.

4. Creating an Ox program with MPI support

There is an example program included in the ParaDiOx package, which is called example.ox. This program code is recommended to be used as a reference for inexperienced MPI programmers on how to use the different MPI functions. This manual will be based on that example and will explain the different functions in detail. The example is made as a suggestion only on how you may want to make use of the distributing power of MPI – it is worth mentioning that there are many more advanced possibilities available (e.g. letting clients communicate with each other etcetera). The example is simple and straightforward in order to make it (relatively) easy to understand.

4.1. Function reference

We have chosen to list each function implemented as of this date. As stated in the next section, there are no limits on increasing the functionality of the dll file connecting the Ox environment with the MPI standard. Each function is described in detail how to use it, what arguments it needs and a brief example code at the end of each summary. As said, if you want to see the functions in actual use, the example source code is available with this ParaDiOx release. The examples in this user's manual are, of course, in Ox code.

4.1.1. MPI_Init()

This function is used to initialize the MPI connections on all participating computers. It should often be used quite early in the Ox program, since every process must communicate with other processes. *Note: the MPI standard can have arguments in the init function. However, the way implemented in the ParaDiOx release, this has been simplified just to do the basic necessary initializations.*

```
MPI_Init(); //as simple as that
```

4.1.2. MPI_Comm_Size()

This function returns an integer, which represent the number of processes available in the domain. It can be used if one wants to loop over all processes, since it is an easy way of knowing the upper limit of the loop sequence.

```
decl i, j;
i = MPI_Comm_Size(); //assuming MPI_Init etc
for(j = 0; j < i; j++) {
    //do stuff here
}
```

4.1.3. MPI_Comm_Rank()

This function returns an integer, which is the unique number (rank) of the calling process. It can be used to find out whether the running process is running as root or as a slave (if you use that kind of hierarchy).

```
decl myId;
myId = MPI_Comm_Rank(); //assuming MPI_Init etc…
if(myId == 0)
    //do master stuff
else
    //do non-master stuff
```

4.1.4. MPI_Get_processor_name(String *name)

This function is used for finding out what name the processor has. Since a String is involved, one cannot use the same return methodology as with integers, since the c-language does not support that. Instead you need to have the address to the name as an argument. This way the function in the dll file will receive that address and then do what needs to be done and then stores the name on that address. This way, the value of the ”String” used as an argument has been changed into the processor name!

```
decl processor_name;
MPI_Get_Processor_name(&processor_name); //assuming MPI_Init etc…
print(processor_name); //Ox will print the processor name fetched via
                       //MPI
```

4.1.5. MPI_Bcast(const *message, int root)

This function is used for sending a message to all processes, hence the name broadcast. The message is sent using the same technique as the MPI_Get_Processor_name function; the address to the String is sent as an argument. In addition to this, an integer is sent as an argument. This number should be used as the sender's Id number, which was received by the MPI_Comm_Rank function above.

The idea is that each process (sender or receiver) uses the same function. If the specified integer is the same as the actual process – then the process sends the broadcast message. If on the other hand the specified integer is not the same as the calling process, it will instead fetch the broadcasted message and thus receiving it!

The message can be of different types. We have implemented the following: int, double, matrix, String and Array. In all cases, all you have to do is to send the address to the declared variable. If other types are needed to be sent, that type needs to be defined in the sending function in the dll file.

```
//assuming same myId declaration from example code above
decl message;
if(myId == 0)
{
    message = 1234;  //defines the sending message as that integer
    MPI_Bcast(&message, 0);  //assuming MPI_Init etc
}
else
{
    MPI_Bcast(&message, 0);  //assuming MPI_Init etc
    print(message);  //receives the message from sending process and
                     // prints
}
```

4.1.6. MPI_Reduce(String *sendbuf, String *receivebuf, String operation, int root)

This function is used for "reducing" arguments into one, using the specified operation. If the desired goal is to add the integers that the slaves have, they can all call the reduce function with the address to its integer, specifying the same result address. The idea is that the MPI function will then add all these elements and store the result at that address and send it to the specified recipitant (most often the root master). For example, the code could be:

```
decl i = 4;
decl answer = 0;
char* operation = "MPI_SUM";
MPI_Reduce(&i, &answer, &operation, 0);
```

As of this date, the dll can handle the following different types of data:

int, double, ox_matrix, ox_string and ox_array.

Also, there are several different types of operations supported in the current release. They are: `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_BOR, MPI_LOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC and MPI_MINLOC.`

4.1.7. MPI_Finalize()

This function is used for finalizing the MPI connection before ending the program. It is closely connected to the MPI_Init function, since that one is used in the same way for setting up the connection.

```
MPI_Finalize(); //as simple as that
```

4.1.8. MPI_Wtime()

This function is used for keeping track of how long time something takes in a process.

```
decl startwtime = 0.0, endwtime = 0.0, wtime; //MPI_Wtime uses doubles
startwtime = MPI_Wtime(); //assuming MPI_Init etc
```

```
//do some time demanding stuff
endwtime = MPI_Wtime();
time = endwtime – startwtime; //assuming MPI_Init etc
print(“\nTime taken to do the stuff: “, time);
```

4.1.9. MPI_Send(const *sendbuf, int target, int tag)

This function is used for sending text messages between processes. The idea with sending the address of the String (as in the broadcast function) is used here. The other two arguments define the identity of the specific receiver and an extra number, which serves as an information tag to the receiver.

The different types implemented are: int, double, matrix, string and array. As with the broadcast function you have to update the dll file if you need to send another type of message.

```
decl message = 12; //the number to be sent
MPI_Send(&message, 1, 25); //assumes MPI_Init etc. sends mess to process
                           //number 1, with the tag info 25.
```

4.1.10. MPI_Recv(const *receivebuf, int source, int tag)

This function is almost the same as the send function. The only difference is that it receives the message instead and that means that you have to know the source id instead of target id. The tag is used to select the desired message.

```
decl message = 0; //the yet unknown number to be received
MPI_Recv(&message, 0, 25); //assumes MPI_Init etc. receives mess from
                           //process number 0, with the tag info 25.
```

4.1.11. MPI_Probe(int source, int tag)

This function is used to probe the “MPI world” for messages. It is practical to you when you want to check when there is a message out there for you and if it is you can use the MPI_Recv function to receive the message. The MPI_Probe function checks if there is a message from the specified source with that specific tag information and waits until there is such a message. If you just want to check whether there is one or not, you use the other probe function called MPI_Iprobe.

```
decl source = 17, message = 0;
MPI_Probe(source, 26)  // waits until message exists. Assumes MPI_Init
                       // etc.
MPI_Recv(&message, source, 26);
```

4.1.12. MPI_Iprobe(int source, int tag, int *flag)

This function is used to probe the “MPI world” for messages. As stated in the MPI_Probe function, this is used to test whether there is a message or not with the specified source number and tag information. The result is stored in the flag. The flag is an address to an integer, used in the same way as the message variable in the broadcast function. If the flag value is zero, then no new message was found.

```
decl result = 0, source = 17, message = 0;
MPI_Iprobe(source, 26, &result);  //assumes MPI_Init etc. checks source
                                  // (17)
if (result != 0) {
      MPI_Recv(&message, source, 26); //now known that message exist
}
```

5. Support and Services

Very limited support is given on ParaDiOx. This project will be finished in May 2002 and no further improvements are planned. MPICH is still being developed and feel free to recompile ParaDiOx with a newer version of MPICH to access new functionality.

If you have some problems with ParaDiOx you can of course give it a shot and send an email to the project group and *maybe* someone of us have time to help you.