Compute Cluster Server Lab 3: Debugging the parallel MPI programs in Microsoft Visual Studio 2005

One of the most important stages of developing computer programs is a process of searching and fixing bugs that unavoidably appear in complex software systems. In order to simplify and automatize the process (usually called *debugging*) it is necessary to use special software called *debuggers*. In spite of the fact that the complexity of parallel programs usually greatly exceeds the complexity of serial analogues there was lack of qualitative debuggers for parallel programs developed in MPI technology. The newest integrated development environment Microsoft Visual Studio 2005 has an efficient parallel debugger which may be also used for debugging MPI applications.

Lab Objective

The lab objective is to learn how to debug parallel programs in the **Microsoft Visual Studio 2005** environment. We assume that Compute Cluster Server SDK is installed on your workstation and therefore you use MPI implementation of Microsoft Corp. (MS MPI). The lab assignments include:

- Exercise 1 – Run the parallel programs locally,
- Exercise 2 – Debug the parallel programs in Microsoft Visual Studio 2005.

Estimated time to complete this lab: **90 minutes**.

Overview of Parallel Debugging Methods in Microsoft Visual Studio 2005

Debugging (process of searching and fixing bugs) is one of the most important stages of developing computer programs which often takes even more time than the actually writing of the code being debugged. In order to minimize the time for debugging it is necessary to follow the recommendations of the leading software companies and the recognized experts in the field of computer science from the very beginning of software design and coding. For instance, such recommendations require writing automatic tests for all parts of the developed programs and checking values of variables using the **ASSERT** macro. But the procedure of debugging itself must also be done in accordance with recommendations of the highly qualified experts in that field. The right choice of *the debugger* is also of great significance where the debugger is a special program which simplifies essentially the process of searching and fixing bugs, allows running programs in step-by-step mode, checking values of internal variables, setting breakpoints etc. The most important criteria when choosing a debugger is a wide variety of debugging tools and usability. During this Lab we will use the build-in debugger of Microsoft Visual Studio 2005(MS VS), which successfully combines intuitive user interface and a wide spectrum of debugging tools.

One of the most important techniques of debugging is breaking the execution of the program at some specific point of time in order to analyse the values of some program variables. The time of program breaking is defined by selecting so-called *breakpoints* i.e. lines in the source code, on which the execution of the program is interrupted. These breakpoints may be *conditional* if they are supplied with some conditions. In this case the programs are interrupted at the breakpoints if and only if the conditions are true. For example, the condition may be the requirement that the value of some variable has to be greater or equal to the given constant. The right choice of the collection of breakpoint has critical importance for successful debugging. For instance, frequently the analysis of variables just before the program crash will provide enough information to determine the cause of incorrect erroneous work.

Another exceptionally useful tool is the ability to execute the program in step-by-step mode i.e. one line of source code is executed each time you press the **F10** button. At any moment the user can go to the next line or may start to execute the function which is called at the current program line in step-by-step mode as well. Thus, the user can always be at that level of desirable detailing.

In order to estimate all the advantages of debuggers it is necessary to compile your program in a special debugging configuration (which is usually called **Debug**). In this configuration special debugging information is added to the binary files. This information is used by the debugger to visualize the source code of the program being executed.

The other most used tools of Microsoft Visual Studio 2005 are the following:

- The window **Call Stack** – the window shows the current stack of function calls and allows to switch the context to each of functions in the stack (that makes possible to check local variables of the functions),
- The window **Autos**– the window shows values of variables used in the current and the previous lines of the code. Besides the window can show values returned by the called functions. The list of shown variables is formed by MS VS automatically,
- The window **Watch**– the window shows values of variables selected by the user. It makes possible to check variables not presented in the window **Autos**,
- The window **Thread**– the window allows switching from one thread of the process to another,
- The window **Process** – the window allows switching among debugged processes (for example, in the case of MPI programs).

The debugging of parallel MPI programs has some peculiarities caused by the nature of parallel programming. There are several processes working with the same problem simultaneously. And each of the processes might have several command threads. It makes debugging more complicated because in addition to errors being typical for serial programs there are also errors which appear only in parallel ones. One of results of parallel programming is *race conditions* where several processes of the parallel program interact without special synchronization. In that case the sequence of operations depends on the varying state of computing system. So the sequence of the executed operations might be different from one program run to another. Therefore checking of correctness by using tests, one of the most used methods of debugging, can not help as a single test can not uncover race conditions.

However the tools and the methods used in Microsoft Visual Studio 2005 for debugging both serial and parallel programs have many points in common and so if you have some good experience of serial debugging then debugging parallel programs won't be very difficult to you.

As a teaching example of parallel program for the following lab exercises we will use the project developed in the preliminary lab "Parallel Programming Using MPI".

Exercise 1 – Run the parallel programs locally

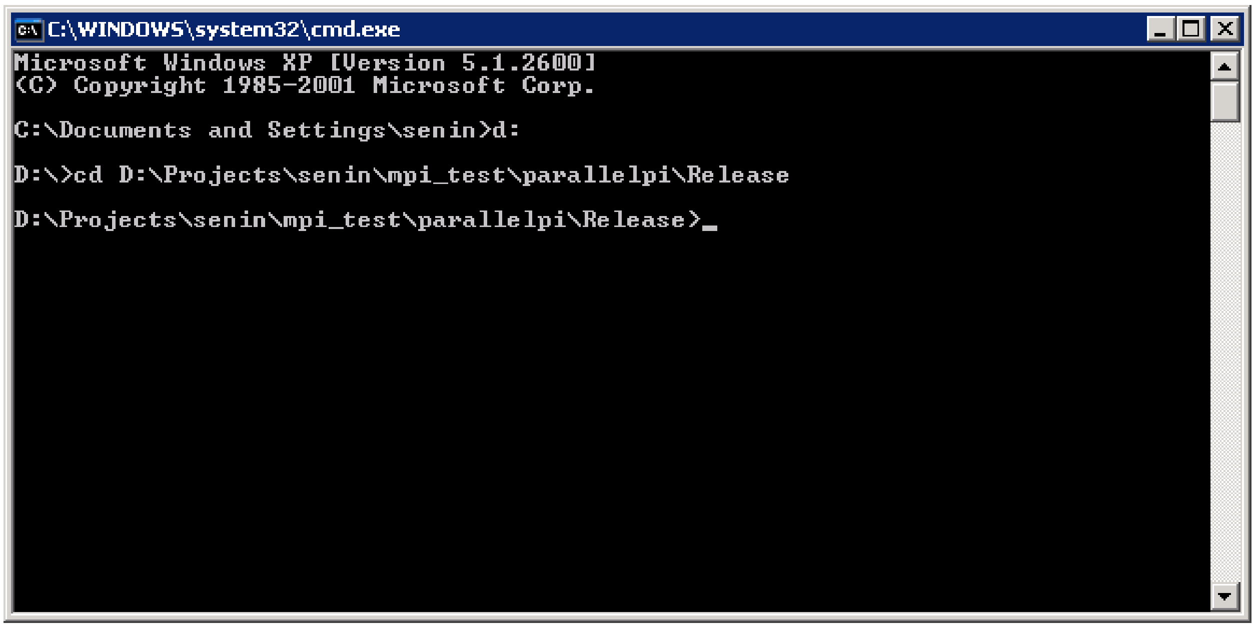
Before starting your program on a multiprocessor computing system (cluster) it is desirable to execute a few tests on a local workstation (in case of using a single-processor computer all processes of the parallel program will run in time-sharing mode). That way of parallel programs execution is usually more effective in respect to debugging as it does not demand any remote access to the cluster and user does not need to wait for actual start of the program on the cluster. Besides such experiments are usually executed for simpler and smaller input parameters. Those all allow to fix quickly the most of bugs in the parallel program.

To execute the experiments locally it is necessary to install both client utilities of Microsoft Compute Cluster Pack and Microsoft Compute Cluster Server SDK.

In this exercise we will learn how to set necessary parameters of Microsoft Visual Studio 2005 for debugging parallel MPI programs and we will start a program in debug mode.

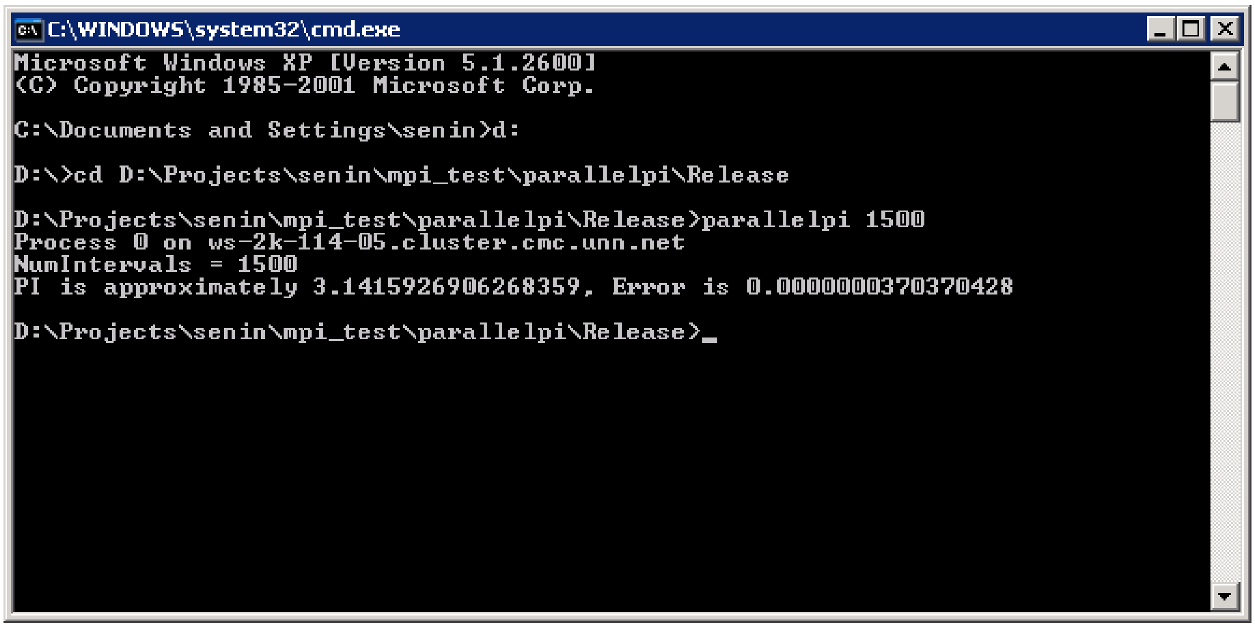
Task 1 –Run the Parallel Program Locally

- Select the configuration **Release** and compile the parallel project of Pi calculation (**parallelpi**) in accordance with instructions in Lab 2 "Carrying out Jobs under Microsoft Compute Cluster Server 2003",
- Open the command window ("**Start->Run**", enter command **cmd** and press the key **Enter**),
- Go to the directory which contains the compiled program (for instance, in order to pass on to the directory "**D:\Projects\senin\mpi_test\parallelpi\Release**" enter a command to set the disk **d:** and then go to the appropriate directory by entering the command "**cd D:\Projects\senin\mpi_test\parallelpi\Release**"),



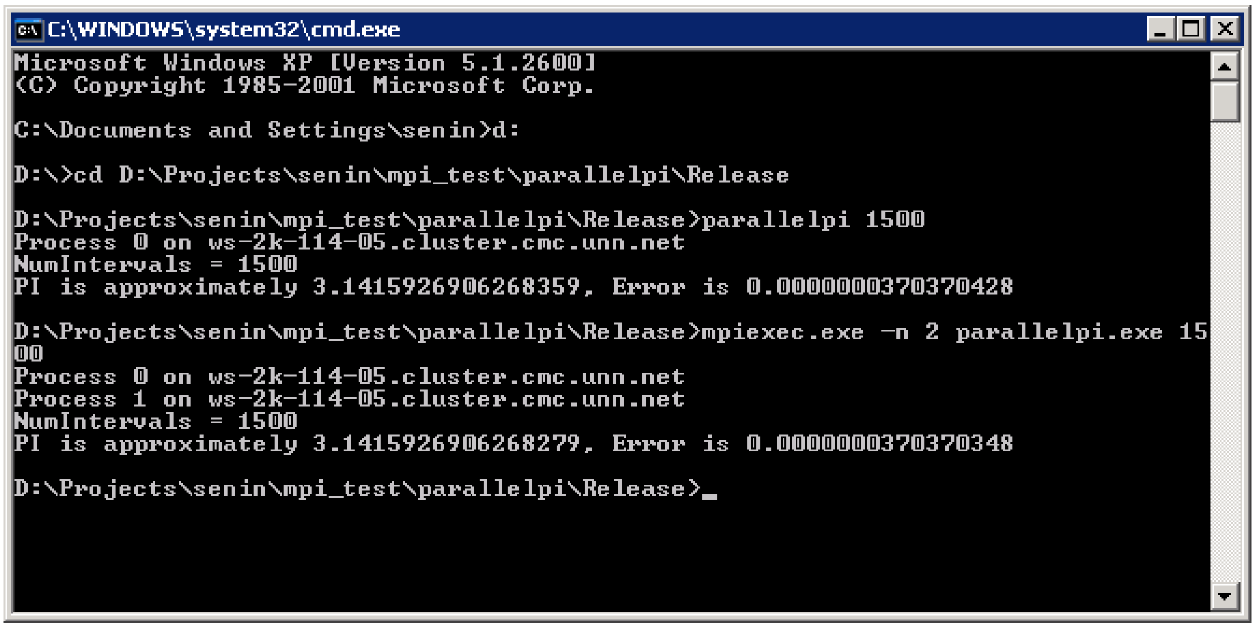
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:

D:\>cd D:\Projects\senin\mpi_test\parallelpi\Release

D:\Projects\senin\mpi_test\parallelpi\Release>_
```

- To start a program in serial mode using 1 process only it is enough to enter the program name and the command line arguments. For instance, "**parallelpi.exe 1500**",



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:

D:\>cd D:\Projects\senin\mpi_test\parallelpi\Release

D:\Projects\senin\mpi_test\parallelpi\Release>parallelpi 1500
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268359, Error is 0.0000000370370428

D:\Projects\senin\mpi_test\parallelpi\Release>_
```

- To start a program in parallel mode on your workstation locally enter the command “**mpiexec.exe –n <number of processes> parallelpi.exe <command line parameters** >”. In our case the only command line parameter of the parallel program of Pi calculation is the number of intervals for definite integral computing,

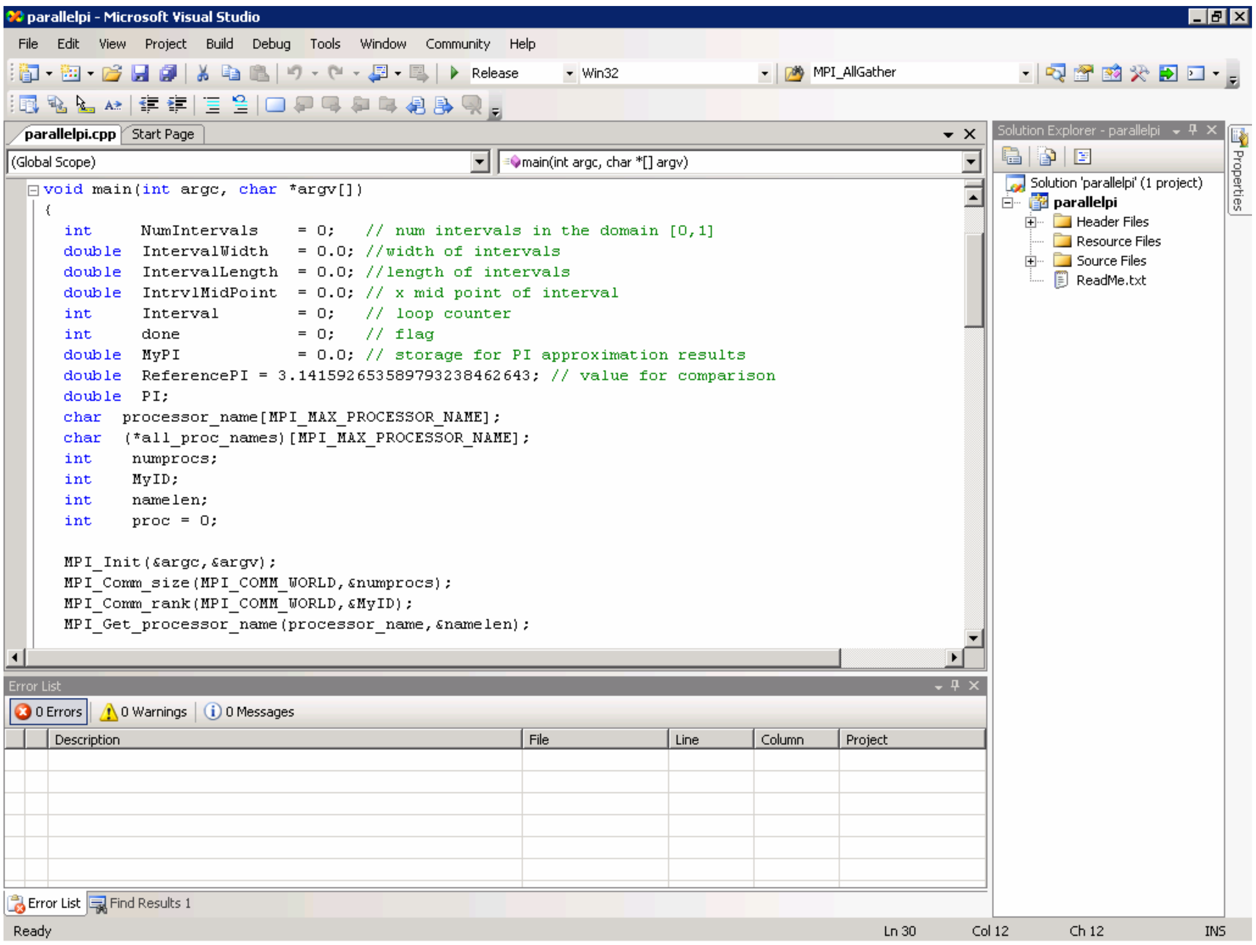


```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\senin>d:

D:\>cd D:\Projects\senin\mpi_test\parallelpi\Release

D:\Projects\senin\mpi_test\parallelpi\Release>parallelpi 1500
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268359, Error is 0.0000000370370428

D:\Projects\senin\mpi_test\parallelpi\Release>mpiexec.exe -n 2 parallelpi.exe 15
00
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
Process 1 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268279, Error is 0.0000000370370348

D:\Projects\senin\mpi_test\parallelpi\Release>_
```

Task 2 – Set the Parallel Debug Mode Parameters

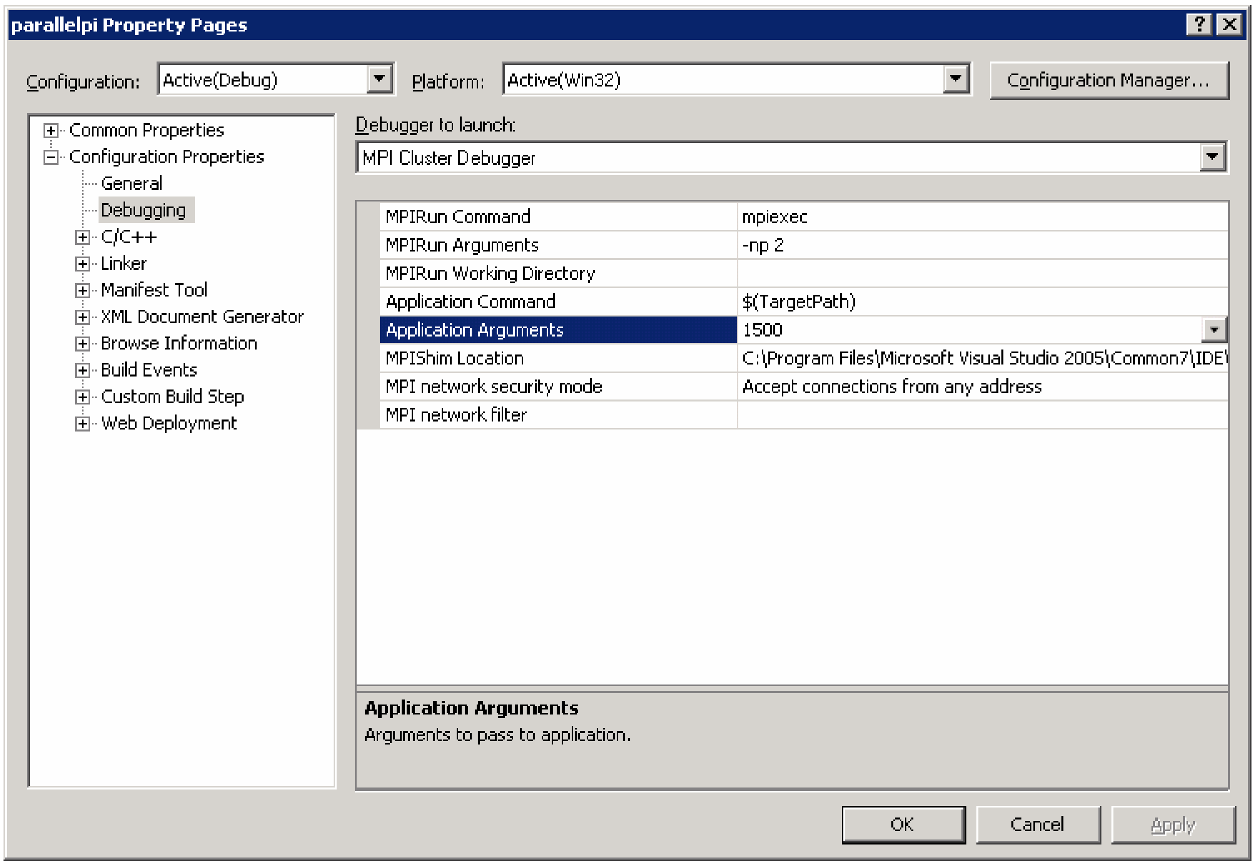
It is necessary to tune up **Microsoft Visual Studio 2005** properly so that you can debug the parallel MPI programs:

- Open the parallel project of Pi computing (**parallelpi**) in Microsoft Visual Studio 2005,



```
void main(int argc, char *argv[])
{
  int      NumIntervals    = 0;  // num intervals in the domain [0,1]
  double   IntervalWidth   = 0.0; //width of intervals
  double   IntervalLength  = 0.0; //length of intervals
  double   IntrvlMidPoint  = 0.0; // x mid point of interval
  int      Interval        = 0;  // loop counter
  int      done            = 0;  // flag
  double   MyPI            = 0.0; // storage for PI approximation results
  double   ReferencePI = 3.14159265358979323846264 3; // value for comparison
  double   PI;
  char   processor_name[MPI_MAX_PROCESSOR_NAME];
  char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
  int     numprocs;
  int     MyID;
  int     namelen;
  int     proc = 0;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
  MPI_Get_processor_name(processor_name,&namelen);
```

- Open the menu option **Project->paralellpi Properties…**. In the opened window of project settings choose the item **Configuration Properties->Debugging**. Select the following settings:
  - In the field **Debugger to launch** select **MPI Cluster Debugger**,
  - In the field **MPIRun Command** enter "**mpiexec.exe**" – the name of the program which is used for starting the parallel MPI programs,
  - In the field **MPIRun Argument** enter the arguments of the utility **mpiexec**. For instance, enter "**-np <number of processes>**" to set the number of processes which will be launched,
  - In the field **Application Command** enter the name of the parallel program to start,
  - In the field **Application Argument** enter the command line arguments of the parallel program,
  - In the field **MPIShim Location**" enter the path to the utility "**mpishim.exe**" – a special program delivered with Microsoft Visual Studio 2005. This utility is used for debugging the remote programs,
- Press **OK** to save settings,



parallelpi Property Pages

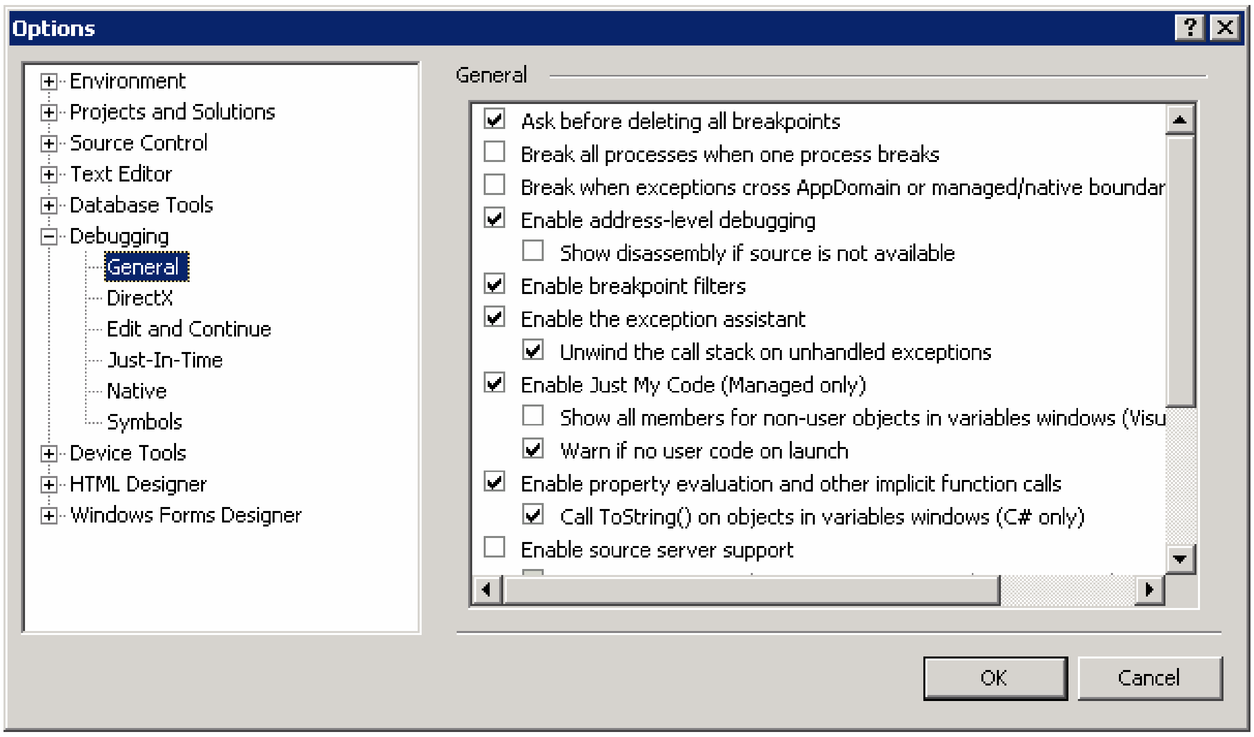
Configuration: Active(Debug) Platform: Active(Win32)

Debugger to launch: MPI Cluster Debugger

MPIRun Command	mpiexec
MPIRun Arguments	-np 2
MPIRun Working Directory	
Application Command	$(TargetPath)
Application Arguments	1500
MPIShim Location	C:\Program Files\Microsoft Visual Studio 2005\Common7\IDE\
MPI network security mode	Accept connections from any address
MPI network filter	

Application Arguments
Arguments to pass to application.

- Choose the menu option **Tools->Options**. In the window of project settings choose the item **Debugging->General**. Check **Break all processes when one process breaks** to break all processes of parallel program in case of one of processes was suspended. If you want the rest processes to be executing when one is suspended uncheck the checkbox (the recommended setting).



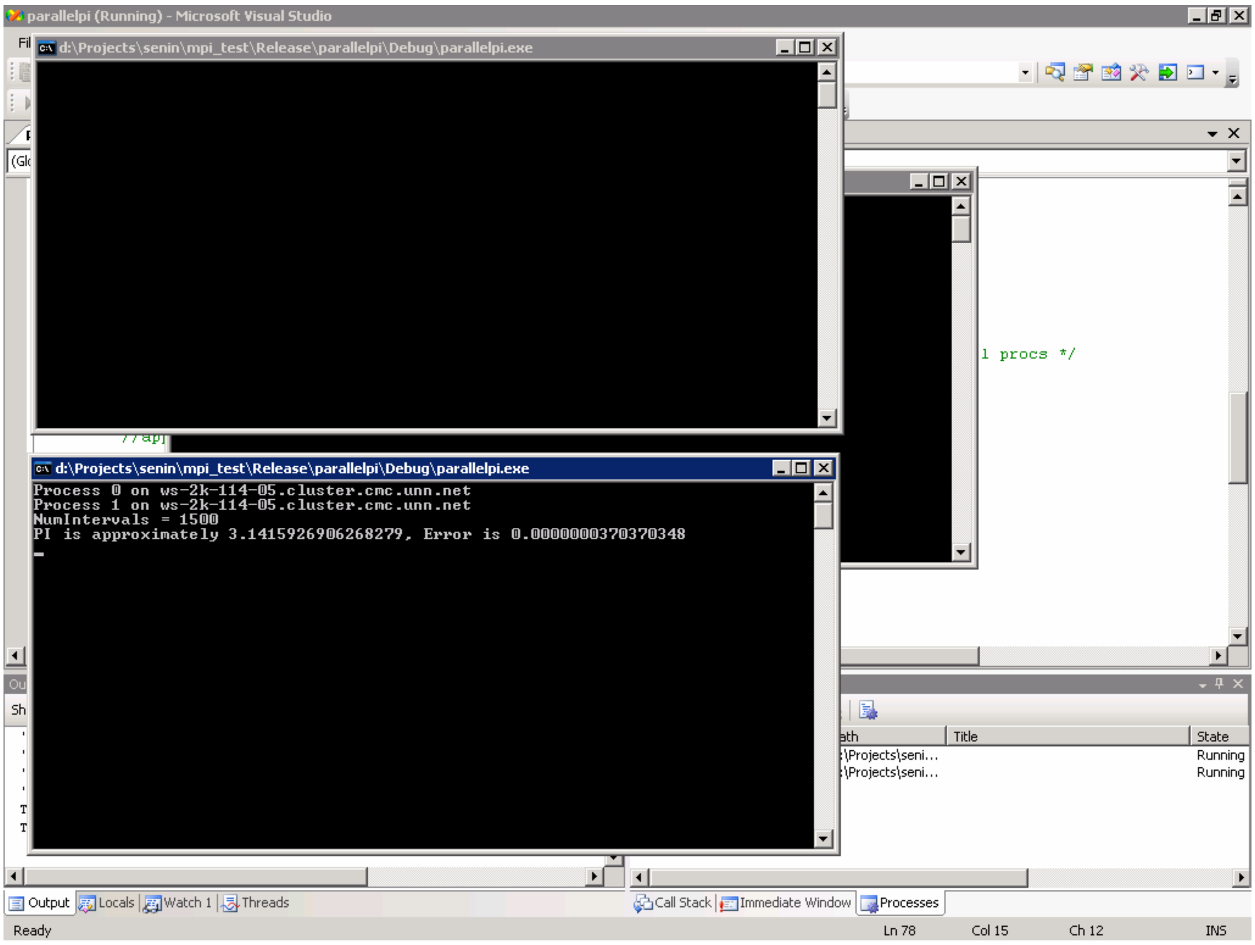
Options

General

- [✓] Ask before deleting all breakpoints
- [ ] Break all processes when one process breaks
- [ ] Break when exceptions cross AppDomain or managed/native boundar
- [✓] Enable address-level debugging
  - [ ] Show disassembly if source is not available
- [✓] Enable breakpoint filters
- [✓] Enable the exception assistant
  - [✓] Unwind the call stack on unhandled exceptions
- [✓] Enable Just My Code (Managed only)
  - [ ] Show all members for non-user objects in variables windows (Visu
  - [✓] Warn if no user code on launch
- [✓] Enable property evaluation and other implicit function calls
  - [✓] Call ToString() on objects in variables windows (C# only)
- [ ] Enable source server support

Task 3 – Run the Parallel Program in Debug Mode

- After settings described in the previous task you will be able to debug the parallel MPI programs. So, if you start the program under Microsoft Visual Studio 2005 (the command **Debug->Start Debugging**) several processes will be started at the same time (the number of processes corresponds to the setting from the previous task). Each of started processes has own console window. You can break each of the processes by using Microsoft Visual Studio 2005 commands and debug it. The detailed description of debugging will be given in the next exercise.

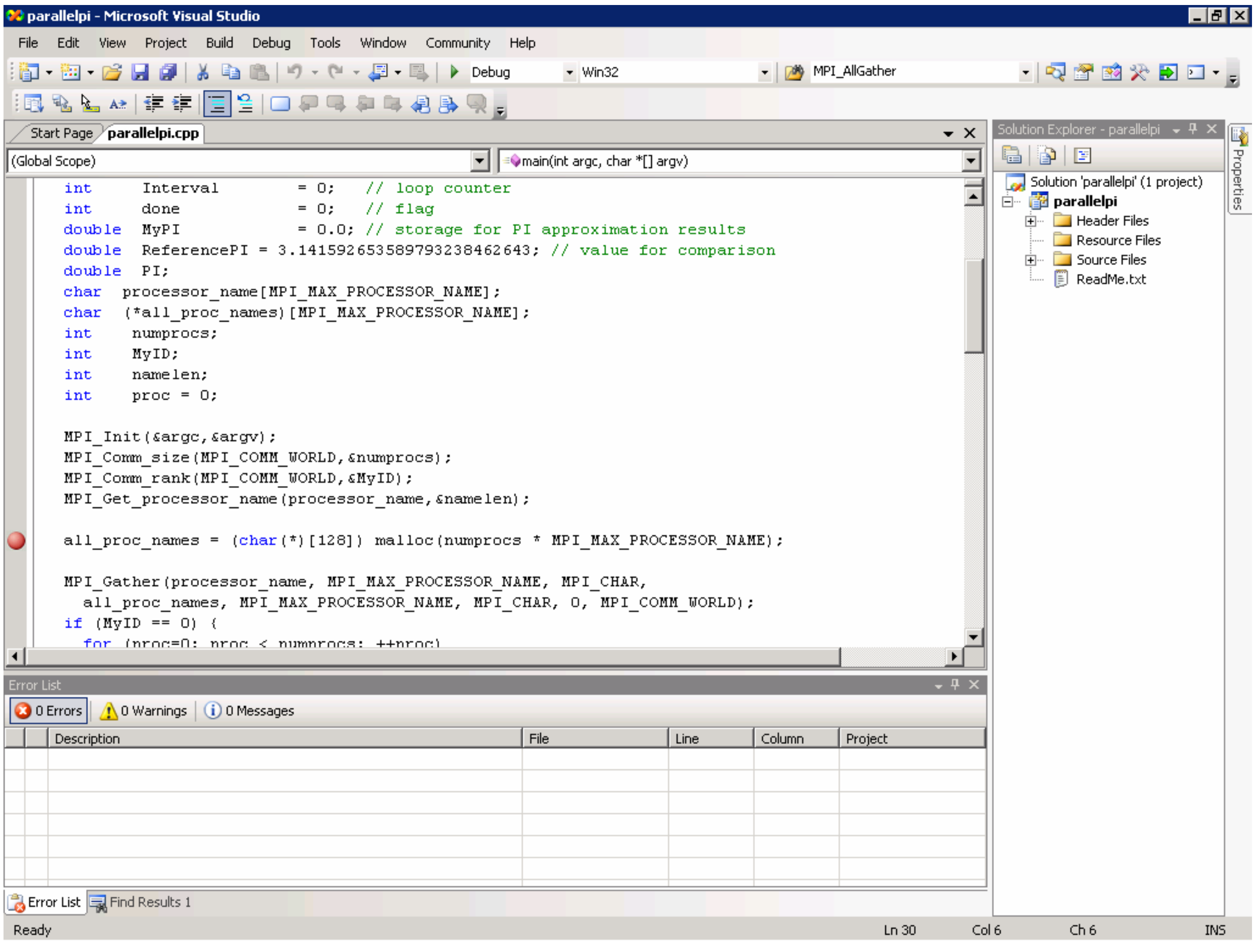

parallelpi (Running) - Microsoft Visual Studio
d:\Projects\senin\mpi_test\Release\parallelpi\Debug\parallelpi.exe
d:\Projects\senin\mpi_test\Release\parallelpi\Debug\parallelpi.exe
Title
State
Running
Running
Output
Locals
Watch 1
Threads
Call Stack
Immediate Window
Processes
Ready
Ln 78
Col 15
Ch 12
INS

```
Process 0 on ws-2k-114-05.cluster.cmc.unn.net
Process 1 on ws-2k-114-05.cluster.cmc.unn.net
NumIntervals = 1500
PI is approximately 3.1415926906268279, Error is 0.0000000370370348
```

Exercise 2 – Debug the parallel programs in Microsoft Visual Studio 2005

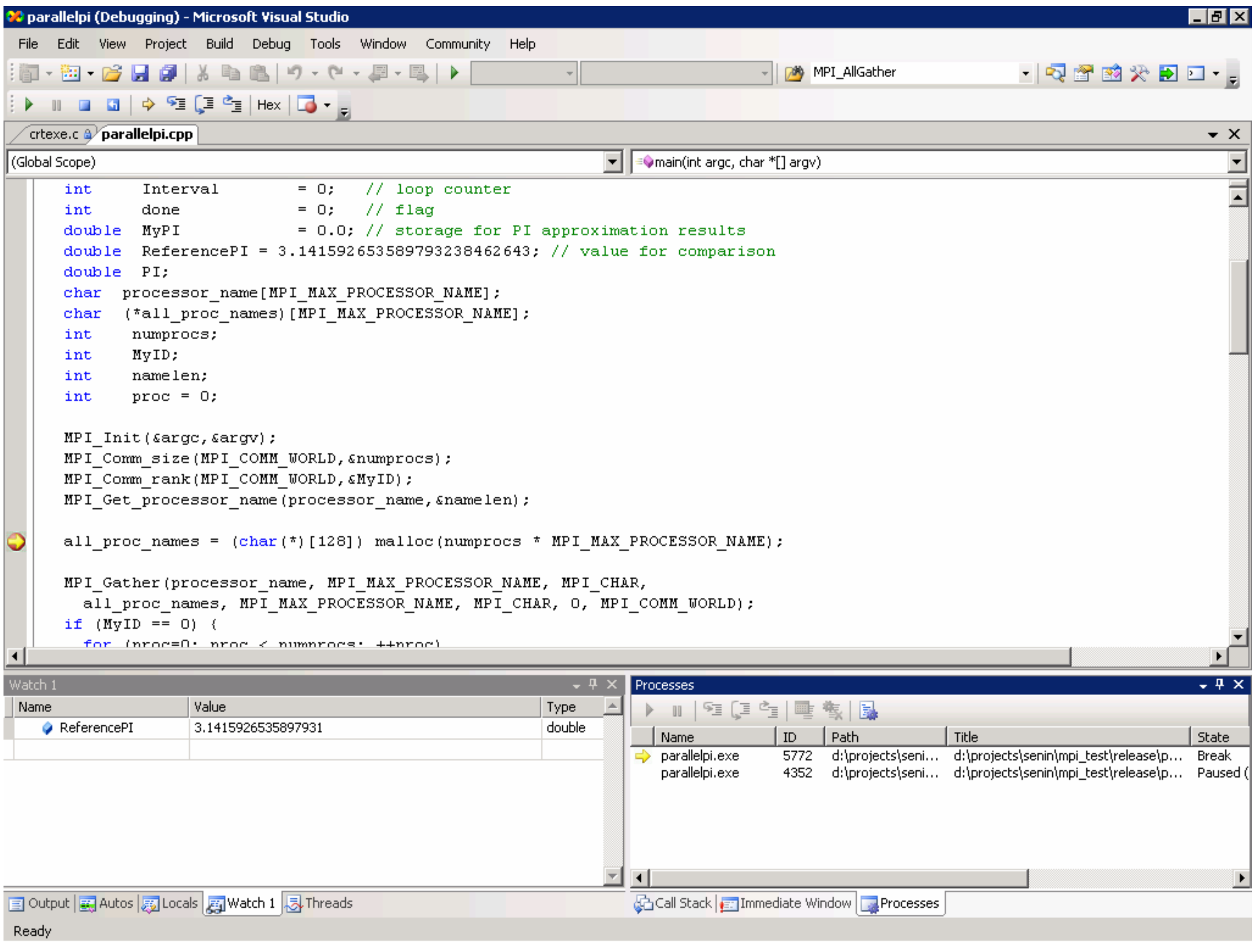
Task 1 – Introduction into the Basic Debugging Methods

- Open the parallel project of Pi calculating (**parallelpi**) and tune up Microsoft Visual Studio 2005 in accordance with the instructions from the previous exercise (if the settings have not been done yet). Set the number of processes to 2. Open the file “**parallelpi.cpp**” by double clicking to its icon in the window **Solution Explorer**,
- Set the breakpoint in the line which allocates the memory by means of using the function **malloc**: set the blinking cursor to the line where you want to set the breakpoint and press the key **F9**,



```
   int      Interval        = 0;    // loop counter
   int      done            = 0;    // flag
   double   MyPI            = 0.0; // storage for PI approximation results
   double   ReferencePI = 3.141592653589793238462643; // value for comparison
   double   PI;
   char   processor_name[MPI_MAX_PROCESSOR_NAME];
   char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
   int     numprocs;
   int     MyID;
   int     namelen;
   int     proc = 0;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
   MPI_Get_processor_name(processor_name,&namelen);

   all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

   MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
     all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
   if (MyID == 0) {
     for (proc=0; proc < numprocs; ++proc)
```

- Select the debugging configuration (**Debug**) in the popup menu on the toolbar. Start the program in debug mode: select the menu option **Debug->Start Debugging** or press the button with green triangle ( ) on the Microsoft Visual Studio 2005 toolbar. The program will be started and three console windows will be opened: the window of the process "**mpiexec.exe**" and 2 console windows corresponding to processes of the parallel program. When the program arrives at the breakpoint the execution of processes will be suspended,

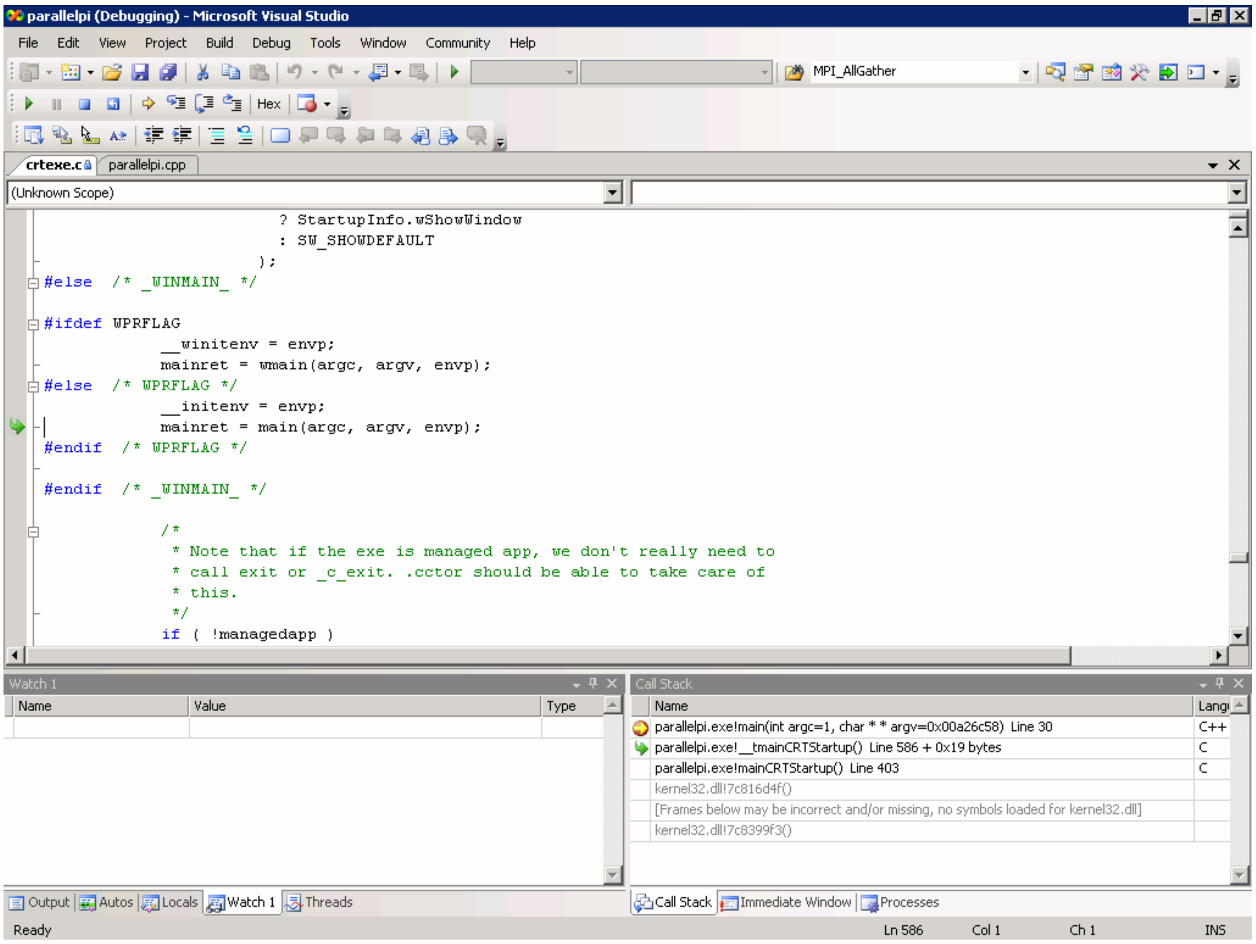

parallelpi (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
MPI_AllGather
Hex
crtexe.c
parallelpi.cpp
(Global Scope)
main(int argc, char *[] argv)
Watch 1
Processes
Output
Autos
Locals
Watch 1
Threads
Call Stack
Immediate Window
Processes
Ready

```
int      Interval       = 0;   // loop counter
int      done           = 0;   // flag
double   MyPI           = 0.0; // storage for PI approximation results
double   ReferencePI = 3.141592653589793238462643; // value for comparison
double   PI;
char   processor_name[MPI_MAX_PROCESSOR_NAME];
char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
int     numprocs;
int     MyID;
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
if (MyID == 0) {
  for (proc=0; proc < numprocs; ++proc)
```

Name	Value	Type
ReferencePI	3.1415926535897931	double

Name	ID	Path	Title	State
parallelpi.exe	5772	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Break
parallelpi.exe	4352	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Paused (

- Open the window **Call Stack** by selecting the menu option **Debug->Windows->Call Stack**. The window shows the current functions stack and allows to switch the context to any function in the stack. It helps to examine a sequence, in which the functions were called and to check the local variables of the functions from the stack. Double click on the function located in the stack under the current one. The context will be changed and you will see the code of a library function which has called the function **main**,

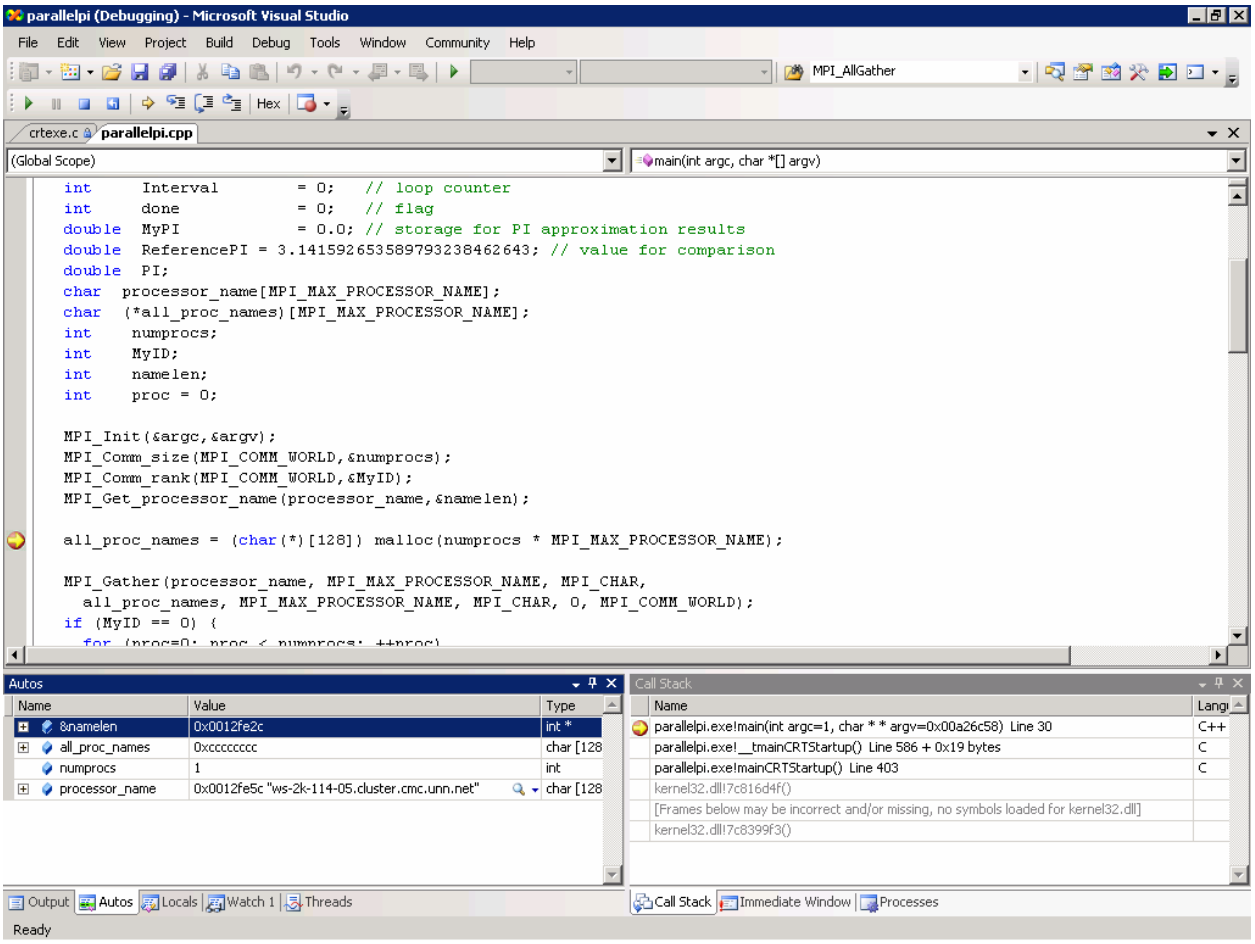

parallelpi (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
MPI_AllGather
Hex
crtexe.c
parallelpi.cpp
(Unknown Scope)

```
                        ? StartupInfo.wShowWindow
                        : SW_SHOWDEFAULT
                      );
#else  /* _WINMAIN_ */

#ifdef WPRFLAG
            __winitenv = envp;
            mainret = wmain(argc, argv, envp);
#else  /* WPRFLAG */
            __initenv = envp;
            mainret = main(argc, argv, envp);
#endif  /* WPRFLAG */

#endif  /* _WINMAIN_ */

            /*
             * Note that if the exe is managed app, we don't really need to
             * call exit or _c_exit. .cctor should be able to take care of
             * this.
             */
            if ( !managedapp )
```

Call Stack

Name	Lang
parallelpi.exe!main(int argc=1, char * * argv=0x00a26c58) Line 30	C++
parallelpi.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
parallelpi.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c816d4f()	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	
kernel32.dll!7c8399f3()	

- Double click on the function **“main”** in the window **Call Stack**. Open the window **Autos by** selecting the menu option **Debug->Windows->Autos**. The window **Autos** shows values of variables used in the current and the previous lines of code. You can not change the set of variables as it is selected by Microsoft Visual Studio 2005 automatically,



parallelpi (Debugging) - Microsoft Visual Studio

crtexe.c | parallelpi.cpp

(Global Scope) | main(int argc, char *[] argv)

```
int      Interval        = 0;   // loop counter
int      done            = 0;   // flag
double   MyPI            = 0.0; // storage for PI approximation results
double   ReferencePI = 3.141592653589793238462643; // value for comparison
double   PI;
char   processor_name[MPI_MAX_PROCESSOR_NAME];
char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
int     numprocs;
int     MyID;
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
if (MyID == 0) {
  for (proc=0; proc < numprocs; ++proc)
```

Autos

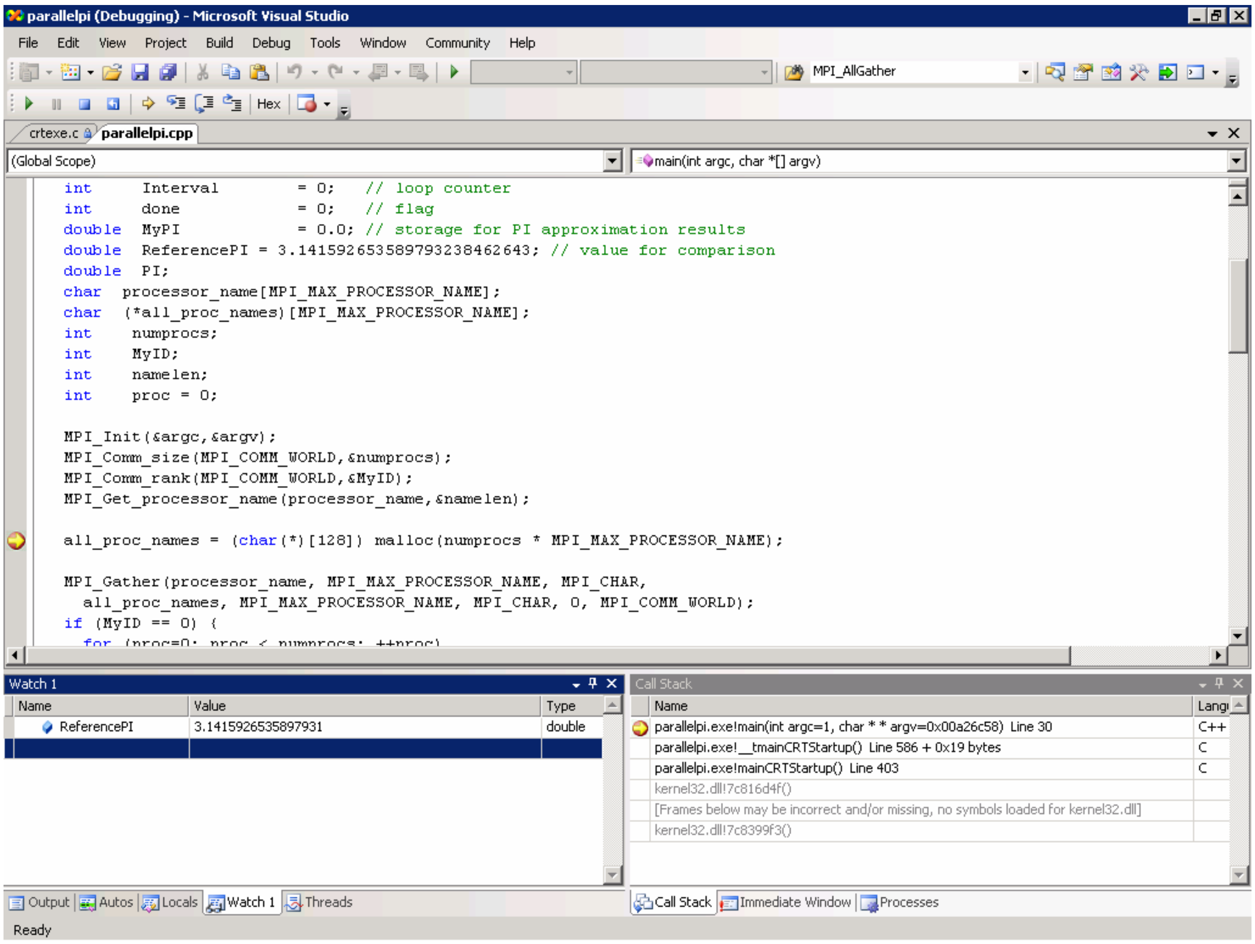
Name	Value	Type
&namelen	0x0012fe2c	int *
all_proc_names	0xcccccccc	char [128
numprocs	1	int
processor_name	0x0012fe5c "ws-2k-114-05.cluster.cmc.unn.net"	char [128

Call Stack

Name	Lang
parallelpi.exe!main(int argc=1, char * * argv=0x00a26c58) Line 30	C++
parallelpi.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
parallelpi.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c816d4f()	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	
kernel32.dll!7c8399f3()	

Ready

- Open the window **Watch** by selecting the menu option **Debug->Windows->Watch->Watch 1**. In this window you can see the values of variables, which are not given in the window **Autos**. For example, enter name of the variable **ReferencePI** in the column **Name** of the window **Watch**, press the key **Enter**. After that the current value of the variable will appear in the column **Value** and its type is shown in the column **Type**,


parallelpi (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
MPI_AllGather
Hex
crtexe.c parallelpi.cpp
(Global Scope)
main(int argc, char *[] argv)

```
int      Interval        = 0;    // loop counter
int      done            = 0;    // flag
double   MyPI            = 0.0; // storage for PI approximation results
double   ReferencePI = 3.141592653589793238462643; // value for comparison
double   PI;
char   processor_name[MPI_MAX_PROCESSOR_NAME];
char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
int     numprocs;
int     MyID;
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
if (MyID == 0) {
  for (proc=0; proc < numprocs; ++proc)
```

Watch 1

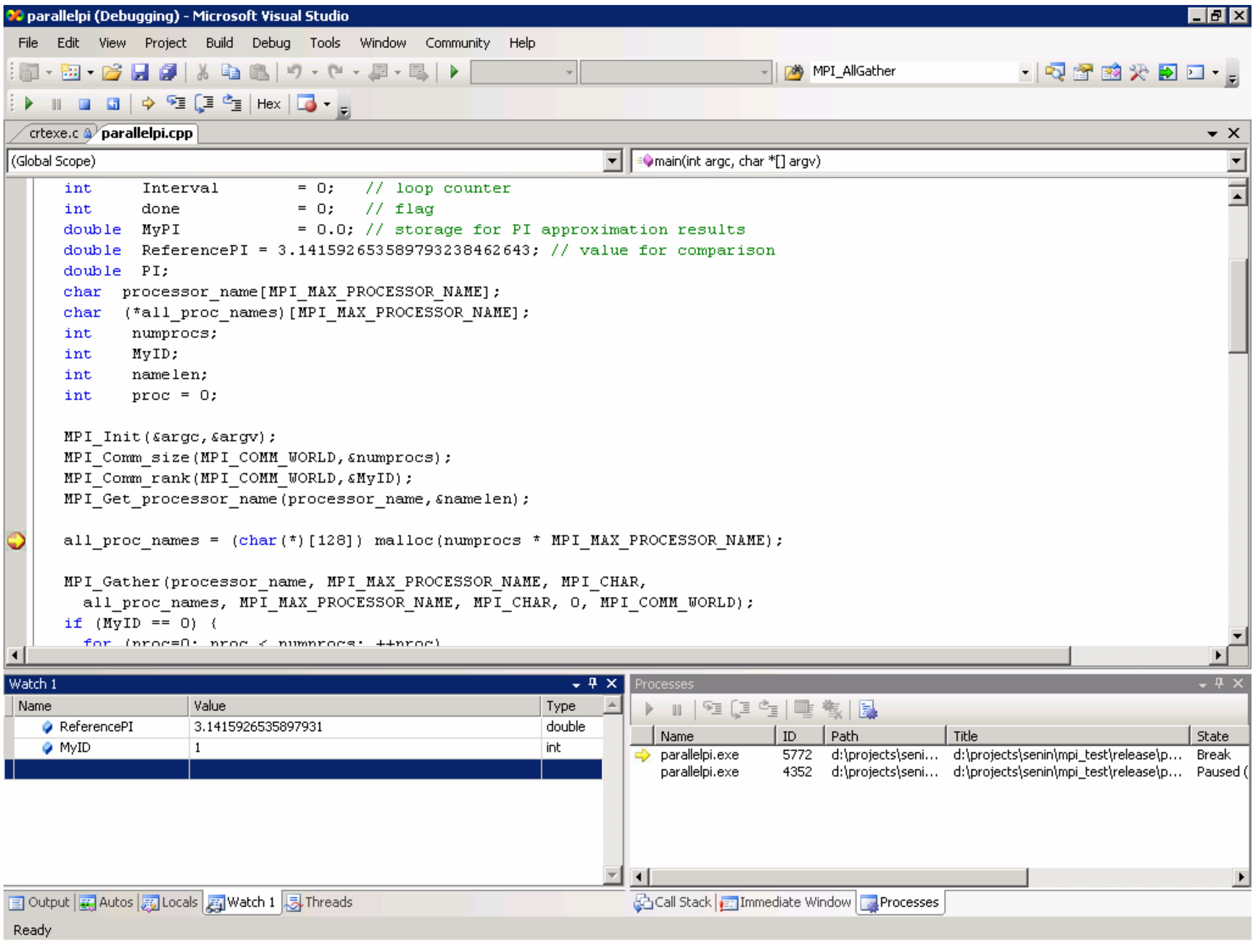
Name	Value	Type
ReferencePI	3.1415926535897931	double

Call Stack

Name	Lang
parallelpi.exe!main(int argc=1, char * * argv=0x00a26c58) Line 30	C++
parallelpi.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
parallelpi.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c816d4f()	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	
kernel32.dll!7c8399f3()	

Output Autos Locals Watch 1 Threads
Call Stack Immediate Window Processes
Ready

- Open the window **Processes** by selecting the menu option **Debug->Windows->Processes**. You will see the window with the list of processes of the parallel MPI program (2 processes in our case). Input the variable **MyID** (the identifier of the current process) in the window **Watch**. Keep in mind the variable value. After that change the current process by double click on another process in the window **Processes**. Watch the value of the variable **MyID** which should be different from the previous one as each process in the MPI program has a unique identifier. While using the window **Processes** it is necessary to take into account two aspects:
  - You can activate only a suspended process (to suspend the process you can set a breakpoint or select the running process in the window **Processes** and execute the command **Break Process** using the button with the icon in the form of two vertical lines ⏸ in the window **Processes**
  - There is the column **ID** in the window **Processes** - the Windows process identifier. It is important to remember that the mentioned identifier is not related to the identifier (rank) of the process in MPI program,



parallelpi (Debugging) - Microsoft Visual Studio

crtexe.c | parallelpi.cpp

(Global Scope) | main(int argc, char *[] argv)

```
int      Interval         = 0;    // loop counter
int      done             = 0;    // flag
double   MyPI             = 0.0; // storage for PI approximation results
double   ReferencePI = 3.141592653589793238462643; // value for comparison
double   PI;
char   processor_name[MPI_MAX_PROCESSOR_NAME];
char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
int     numprocs;
int     MyID;
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
if (MyID == 0) {
  for (proc=0; proc < numprocs; ++proc)
```

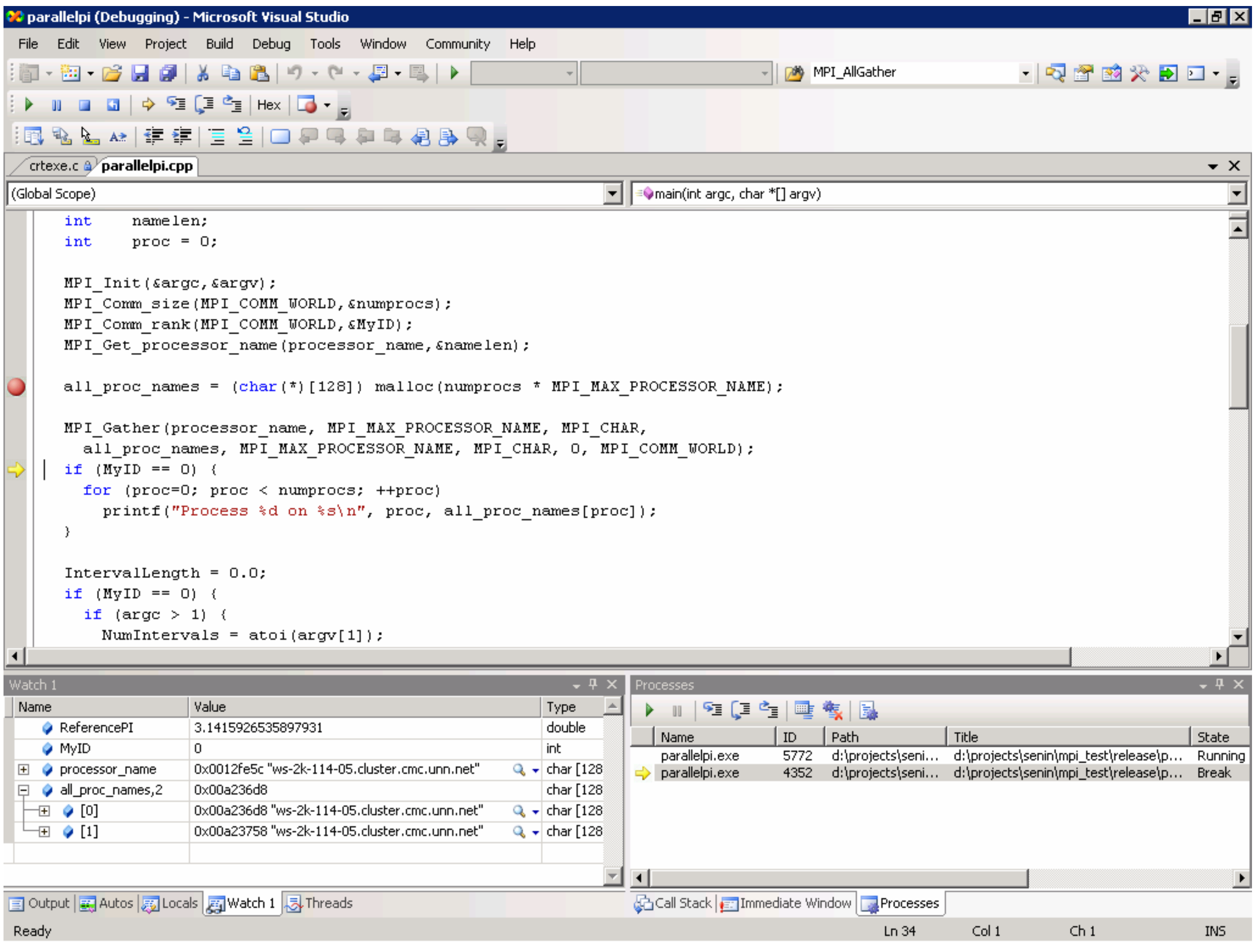
Watch 1

Name	Value	Type
ReferencePI	3.1415926535897931	double
MyID	1	int

Processes

Name	ID	Path	Title	State
parallelpi.exe	5772	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Break
parallelpi.exe	4352	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Paused (

- Select the process with the identifier **MyID** equal to 0. Input the variables **processor_name** and **all_proc_names** in the window **Watch**. The variable **all_proc_names** is an array of node names on which parallel processes have been started up. The array is only used on the process with the identifier 0. To visualize the values of two array elements enter the text "**all_proc_names,2**" in the column **Name**. After that you may see the values of first two array elements pressing "+" on the left of the variable name. Keep pressing the button **F10** for the step-by-step execution of the program. After each step you may see changing of the values of the program variables. So, after the function call **MPI_Gather** the process with the identifier 0 will contain the names of all the nodes on which parallel processes are executed. If the current code line is a function call we can start to examine the execution of this function by pressing the key **F11** (in our example all the functions being called are system ones and you will not be able to see their source code).


parallelpi (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
MPI_AllGather
Hex
crtexe.c
parallelpi.cpp
(Global Scope)
main(int argc, char *[] argv)
Watch 1
Processes
Output Autos Locals Watch 1 Threads
Call Stack Immediate Window Processes
Ready Ln 34 Col 1 Ch 1 INS

```
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
if (MyID == 0) {
  for (proc=0; proc < numprocs; ++proc)
    printf("Process %d on %s\n", proc, all_proc_names[proc]);
}

IntervalLength = 0.0;
if (MyID == 0) {
  if (argc > 1) {
    NumIntervals = atoi(argv[1]);
```

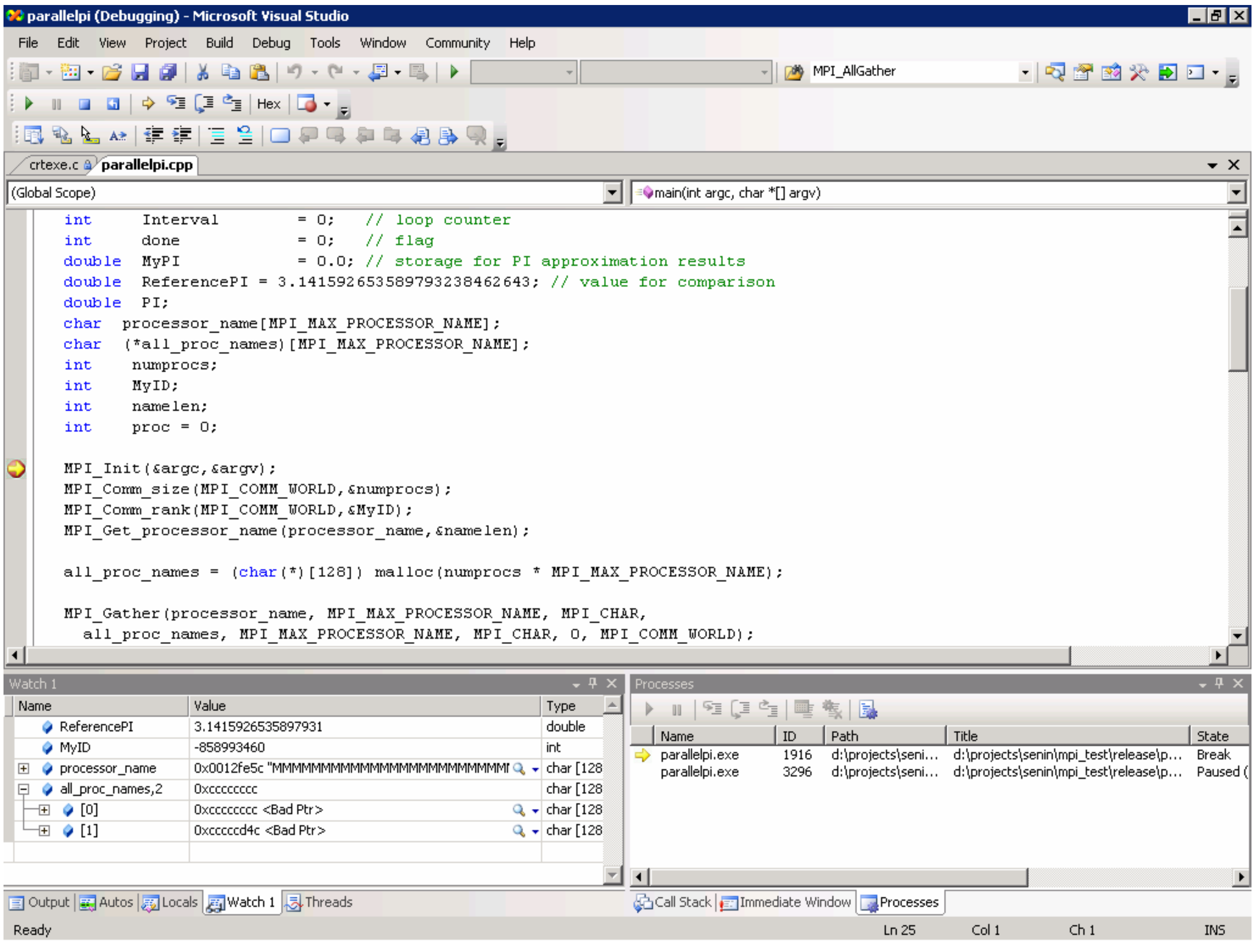
Name	Value	Type
ReferencePI	3.1415926535897931	double
MyID	0	int
processor_name	0x0012fe5c "ws-2k-114-05.cluster.cmc.unn.net"	char [128
all_proc_names,2	0x00a236d8	char [128
[0]	0x00a236d8 "ws-2k-114-05.cluster.cmc.unn.net"	char [128
[1]	0x00a23758 "ws-2k-114-05.cluster.cmc.unn.net"	char [128

Name	ID	Path	Title	State
parallelpi.exe	5772	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Running
parallelpi.exe	4352	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Break

Task 2 – Using the Conditional Breakpoints

We have already used breakpoints in the previous tasks. In this task we will consider how to use conditional breakpoints, i.e. breakpoints which are supplied with some conditions and the program execution is suspended at breakpoint lines only in cases when the breakpoint conditions are true. Using conditional breakpoints is more efficient in several cases as a programmer can set necessary requirements of interrupting the program execution in detail:

- Open the parallel project of Pi calculation (**parallelpi**) and make settings necessary for debugging MPI programs (if it has not been done yet) and choose the number of the started processes equal to 2. Open the file **parallelpi.cpp** (by double click on the file in the window **Solution Explorer**),
- Set a breakpoint at any line of the program (for instance, in the line with the function **MPI_Init**) and start debugging. After arriving at this breakpoint process execution will be suspended,


parallelpi (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
MPI_AllGather
Hex
crtexe.c
parallelpi.cpp
(Global Scope)
main(int argc, char *[] argv)

```
int      Interval        = 0;   // loop counter
int      done            = 0;   // flag
double   MyPI            = 0.0; // storage for PI approximation results
double   ReferencePI = 3.141592653589793238462643; // value for comparison
double   PI;
char   processor_name[MPI_MAX_PROCESSOR_NAME];
char   (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
int     numprocs;
int     MyID;
int     namelen;
int     proc = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
MPI_Get_processor_name(processor_name,&namelen);

all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

MPI_Gather(processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR,
  all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Watch 1

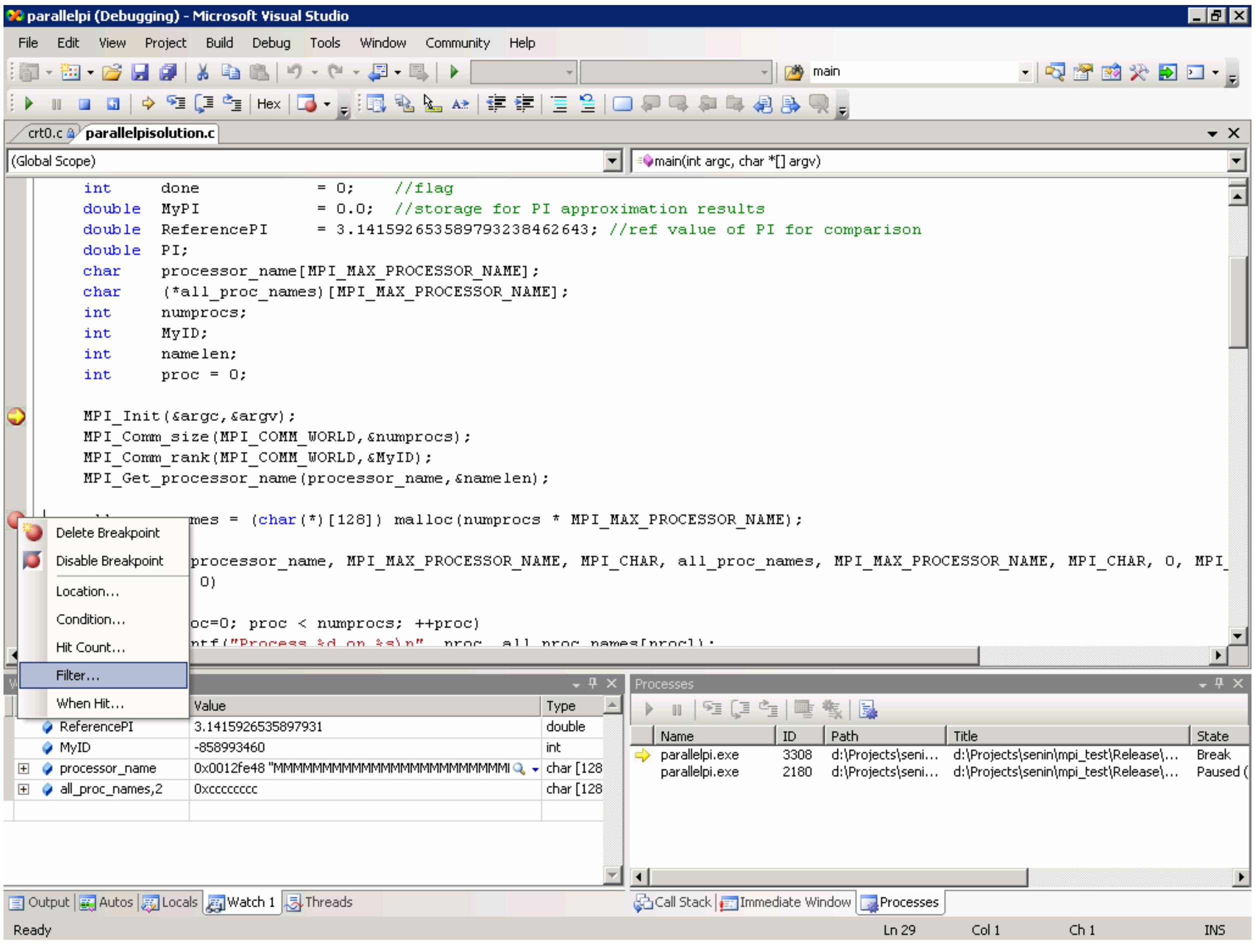
Name	Value	Type
ReferencePI	3.1415926535897931	double
MyID	-858993460	int
processor_name	0x0012fe5c "MMMMMMMMMMMMMMMMMMMMMMMMMM	char [128
all_proc_names,2	0xcccccccc	char [128
[0]	0xcccccccc <Bad Ptr>	char [128
[1]	0xccccccd4c <Bad Ptr>	char [128

Processes

Name	ID	Path	Title	State
parallelpi.exe	1916	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Break
parallelpi.exe	3296	d:\projects\seni...	d:\projects\senin\mpi_test\release\p...	Paused (

Output Autos Locals Watch 1 Threads
Call Stack Immediate Window Processes
Ready Ln 25 Col 1 Ch 1 INS

- You may specify breakpoints which will interrupt only the processes (or threads) you need. To do it set another breakpoint below the current program line, click on the red sign of the new breakpoint by the right mouse button and choose the option **Filter** in the context menu,


parallelpi (Debugging) - Microsoft Visual Studio

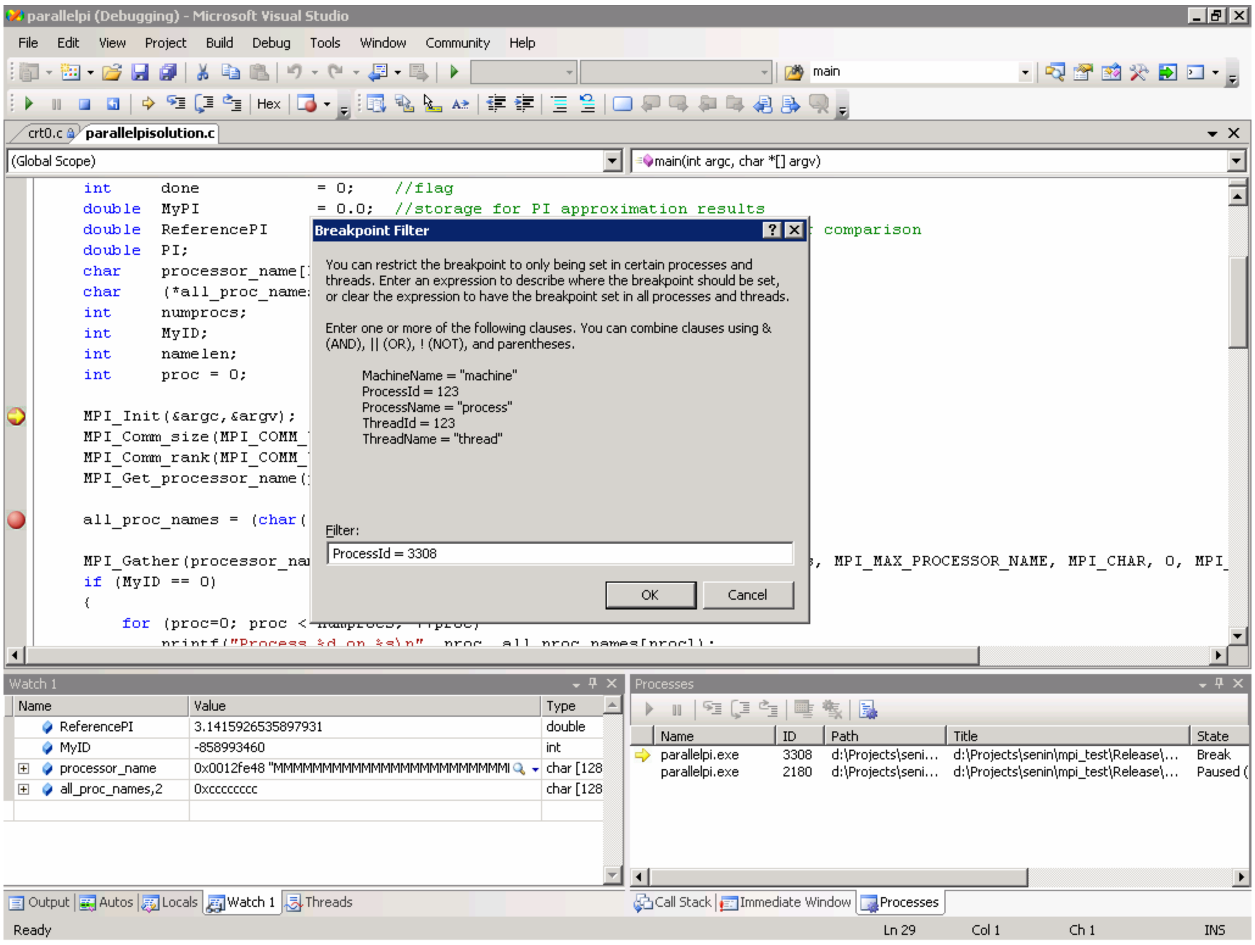
```
    int     done              = 0;    //flag
    double  MyPI              = 0.0;  //storage for PI approximation results
    double  ReferencePI       = 3.141592653589793238462643; //ref value of PI for comparison
    double  PI;
    char    processor_name[MPI_MAX_PROCESSOR_NAME];
    char    (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
    int     numprocs;
    int     MyID;
    int     namelen;
    int     proc = 0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
    MPI_Get_processor_name(processor_name,&namelen);

              mes = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

              processor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, all_proc_names, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_
               0)

              oc=0; proc < numprocs; ++proc)
```

	Value	Type
ReferencePI	3.1415926535897931	double
MyID	-858993460	int
processor_name	0x0012fe48 "MMMMMMMMMMMMMMMMMMMMMMMMMI	char [128
all_proc_names,2	0xcccccccc	char [128

Processes

Name	ID	Path	Title	State
parallelpi.exe	3308	d:\Projects\seni...	d:\Projects\senin\mpi_test\Release\...	Break
parallelpi.exe	2180	d:\Projects\seni...	d:\Projects\senin\mpi_test\Release\...	Paused (

- In the open window you can indicate those processes and threads execution of which must be interrupted when arriving at the breakpoint chosen. Moreover, it is possible to indicate the computational node name on which the breakpoint will interrupt processes (this parameter is used in the case if a subset of the processes to be debugged is running on remote nodes). While writing logical expressions you can use logical operations **AND** (you can also use the sign **&**), **OR** (||), **NOT** (**!**). For instance, to indicate that single process should be interrupted, the expression **ProcessId = <process identifier>** should be used where process identifier may be found out in the window **Processes**. Press the button **OK** to save the settings,



parallelpi (Debugging) - Microsoft Visual Studio

```
    int      done            = 0;    //flag
    double   MyPI            = 0.0;  //storage for PI approximation results
    double   ReferencePI                                  comparison
    double   PI;
    char     processor_name[
    char     (*all_proc_name
    int      numprocs;
    int      MyID;
    int      namelen;
    int      proc = 0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_
    MPI_Comm_rank(MPI_COMM_
    MPI_Get_processor_name(

    all_proc_names = (char(

    MPI_Gather(processor_na                               , MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, MPI_
    if (MyID == 0)
    {
        for (proc=0; proc <
            printf("Process %d on %s\n", proc, all_proc_names[proc]);
```

Breakpoint Filter

You can restrict the breakpoint to only being set in certain processes and threads. Enter an expression to describe where the breakpoint should be set, or clear the expression to have the breakpoint set in all processes and threads.

Enter one or more of the following clauses. You can combine clauses using & (AND), || (OR), ! (NOT), and parentheses.

MachineName = "machine"
ProcessId = 123
ProcessName = "process"
ThreadId = 123
ThreadName = "thread"

Filter: ProcessId = 3308

Watch 1

Name	Value	Type
ReferencePI	3.1415926535897931	double
MyID	-858993460	int
processor_name	0x0012fe48 "MMMMMMMMMMMMMMMMMMMMMMMMMI	char [128
all_proc_names,2	0xcccccccc	char [128

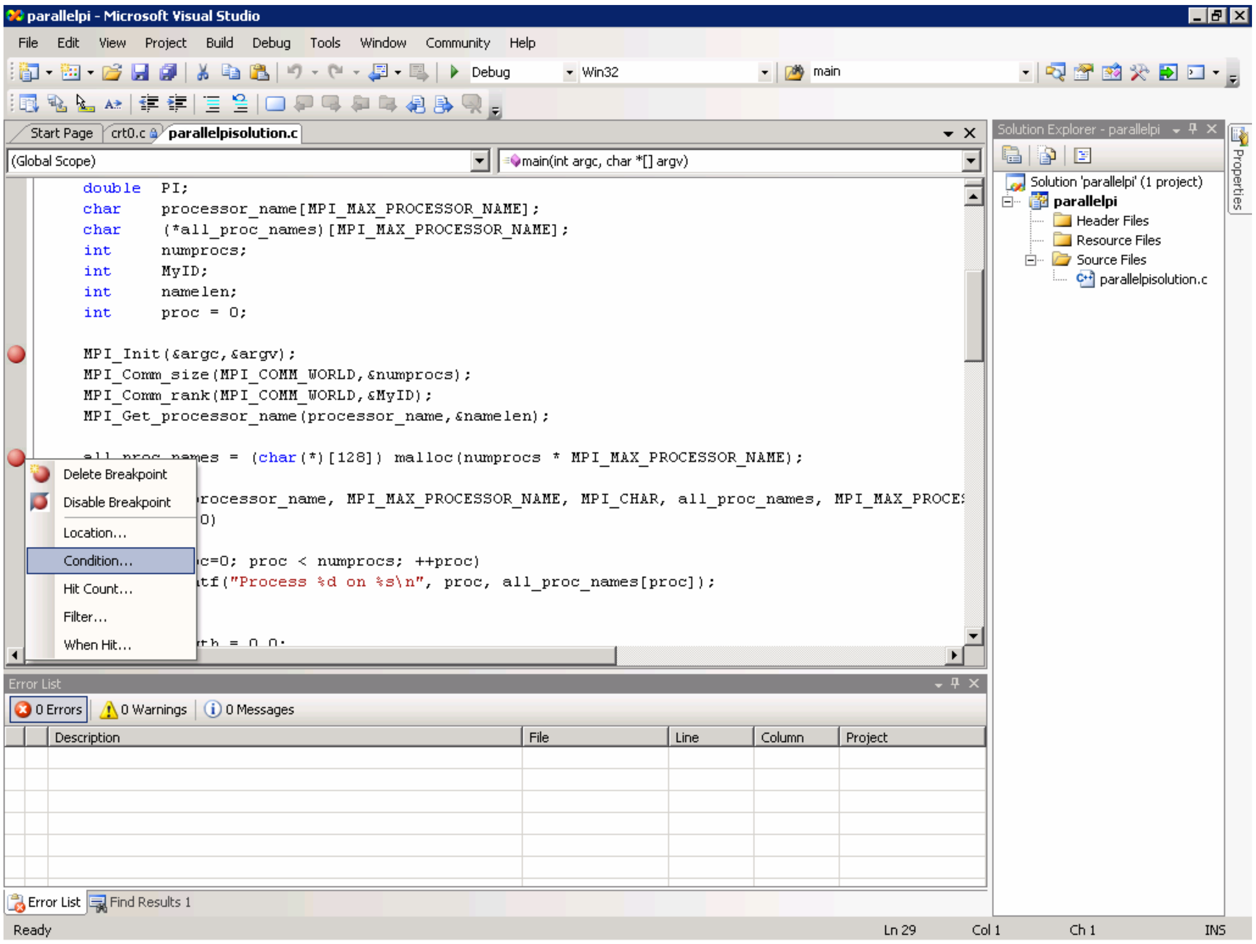
Processes

Name	ID	Path	Title	State
parallelpi.exe	3308	d:\Projects\seni…	d:\Projects\senin\mpi_test\Release\…	Break
parallelpi.exe	2180	d:\Projects\seni…	d:\Projects\senin\mpi_test\Release\…	Paused (

Ready   Ln 29   Col 1   Ch 1   INS

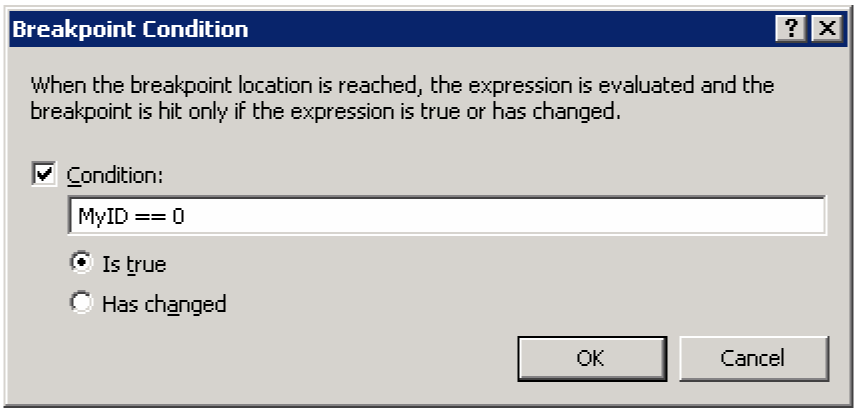
File Edit View Project Build Debug Tools Window Community Help
main
Hex
crt0.c
parallelpisolution.c
(Global Scope)
main(int argc, char *[] argv)
OK
Cancel
Output Autos Locals Watch 1 Threads
Call Stack Immediate Window Processes

- Moreover, you may set conditions for values of the variables. To enter the conditions right-click on the breakpoint and choose the option **Conditions**,



```
    double   PI;
    char     processor_name[MPI_MAX_PROCESSOR_NAME];
    char     (*all_proc_names)[MPI_MAX_PROCESSOR_NAME];
    int      numprocs;
    int      MyID;
    int      namelen;
    int      proc = 0;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
    MPI_Get_processor_name(processor_name,&namelen);

    all_proc_names = (char(*)[128]) malloc(numprocs * MPI_MAX_PROCESSOR_NAME);

                rocessor_name, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, all_proc_names, MPI_MAX_PROCE
                0)

                c=0; proc < numprocs; ++proc)
                tf("Process %d on %s\n", proc, all_proc_names[proc]);
```

- In the opened window you may set conditions of two types:
  - Conditions of the first type allow to interrupt the program execution if the given conditions are true. To use this possibility tick the option **Is true** and enter a conditional expression in the field **Conditions**. While entering the expression the syntax of the algorithmic language C is used. So, for interrupting the process with the identifier 0 only it is necessary to input the condition **MyID == 0**,
  - Conditions of the second type interrupts the program execution if the value of the given expression (not necessarily logical) has changed since previous execution of the statement marked by breakpoint. For switching this type of condition on tick the option **Has changed**,
- Tick the option **Is true** and enter the condition **MyID == 0**. Press the button **OK** to save this condition.



When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

[✓ Condition:] MyID == 0

[✓ Is true] Has changed

Task 3 - Hints for Debugging the Parallel Programs

Microsoft Visual Studio debugger of parallel MPI programs first appeared in Microsoft Visual Studio 2005. However, compiling and debugging the parallel MPI program may also be performed by means of earlier versions of the development environment. In this case the following standard debugging techniques can be applied:

- **Using text messages** which are outputted to a file or on the screen with values of the variables you are interested in and/or information about what part of the program is being executed. Such approach is often called **printf debugging** as the function **printf** is mostly used for message outputting to the console. This method is easy enough because it does not require special skills for working with any debugger and makes possible to find bugs in the course of proper use. Nevertheless, to get the value of a new variable you have to write new code for printing, recompile the program and run the program once again. It requires a lot of time. Moreover, adding a new code to the program may lead to temporary disappearance of some errors,

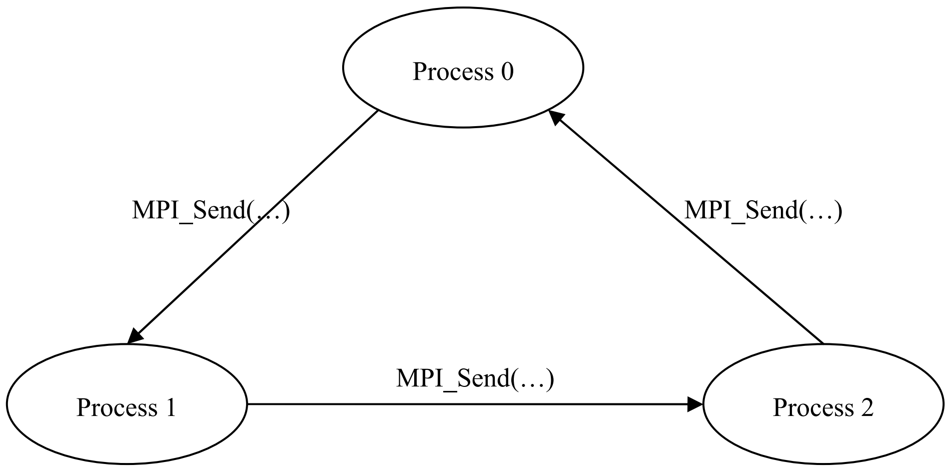
- **Using a serial debugger**. As a MPI program consists of several interacting processes it is possible to start several copies of a serial debugger attaching each copy of the debugger to a certain process of the MPI program. As a rule this approach allows to debug the parallel programs more efficiently than in case of using text messages. The main disadvantage of the approach is the necessity of doing the same repeated actions manually. Let us imagine, for example, the necessity of interrupting 32 processes of the MPI program. If you use the serial debugger you have to select sequentially each of the 32 processes and initiate manually the interruption commands.

The parallel debugger of Microsoft Visual Studio 2005 does not contain such drawbacks and makes possible to save time for debugging. It is achieved by means of the environment considering all the processes of the MPI program as a single program being executed in parallel and thereby provides parallel debugging near to debugging of serial programs. Among the features of the parallel debugging the window **Processes** considered above can be mentioned. This window allows switching among parallel processes. Additional settings of Microsoft Visual Studio 2005 are also useful (see Task 2 of Exercise 1).

Task 4 – Survey of Typical Errors of Parallel Programs

All the errors that appear in the serial programming are also typical for the parallel programming. Nevertheless, there are specific errors arising as the result of availability of several interacting processes executed in parallel. The complexity of the parallel program development results in higher requirements to the skills of programmers which develop parallel programs. Although it is categorize all the types of errors arising in programming with using MPI we will briefly describe the most typical errors making by the beginner developers. We will mention three of the most typical errors in the parallel programming:

- **Deadlock in message transfer**. Let us suppose that *n* processes transmit information one to another in the chain order: the *i*-th process transfers information to the (*i+1*)-th (for indices *i=0,…,n-2*) and the (*n-1*)-th process transmits information to the process with the rank 0. The transmissions are performed with using the function of sending messages *MPI_Send* and the function of receiving messages *MPI_Recv* and each process calls firstly *MPI_Send* and then *MPI_Recv*. These functions are blocking, i.e. *MPI_Send* returns control only after the transmission is finished or after the message was copied to the internal buffer. Thus, the standard assumes the situation when the functions which sends messages returns control only after the transmission is complete. But the transmission can not be finished until the receiving process calls the ***MPI_Recv*** function. In it's part, the function *MPI_Recv* is called only after the sending of the current process messages is finished. As the chain of message transmissions is closed sending the messages can never finish because each process will wait for finishing of sending its message and no one will call the function of message receiving. To avoid such blocking it is possible, for instance, by calling firstly the function ***MPI_Send*** and then the function *MPI_Recv* on the process with even indices, and making these calls in the reverse order on the process with odd indices. In spite of the evidence of this mistake it is often made as in the case of small messages there is a great likelihood that the messages will be placed in the internal memory buffers of the library and the error will not be discovered,



- **Release or change of buffers used for non-blocking transmission.** The non-blocking MPI functions return control immediately and pointers to message arrays are stored in the structures of the MPI library. When the actual message transmission is executed (in a separate thread) the network thread will read the data from the initial message array. Therefore, it is very important that this memory buffer is not deleted or changed before the transmission is finished (you can determine whether the transmission has been completed by means of calling the special functions). Often this rule is forgotten and it leads to unpredictable program behavior,
- **Mismatch of calls for functions of sending and receiving messages.** It is important to pay attention that the function call of sending message corresponds to the function call of receiving message and vice versa. However, in the case when number of transmissions is not known in advance and is determined in the course of computations it is easy to make mistake and not to guarantee the fulfillment of the given condition. This error type includes also calls of functions for sending and receiving messages with mismatched tags of message identification.

Discussions

- How to test the correctness of a parallel program developed with using the MS MPI library locally, before launching on a cluster? What software should be set on your workstation to carry out such testing?
- List and give a short description of the main debug windows in the Microsoft Visual Studio 2005 environment which can be used for debugging.
- What is a breakpoint? Why is it used? What is a conditional breakpoint?
- What is the main advantage of debugging the parallel MPI programs in Microsoft Visual Studio 2005?
- What typical programming errors appear if you use MPI technology?