April 1996

Order Number: 312547-005

Paragon™ System
Interactive Parallel Debugger
Reference Manual

Intel® Corporation

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

286
287
i

i386
i387
i486
i487
i860

Intel
Intel386
Intel387
Intel486
Intel487

iPSC
Paragon

Other brands and names are the property of their respective owners.

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual describes the Paragon™ System Interactive Parallel Debugger (IPD) commands. These commands are available when you are running the debugger. You issue the debugger commands at the IPD prompt.

For a description of how to use the Interactive Parallel Debugger, refer to the *Paragon™ System Application Tools User's Guide*

In this manual, "operating system" refers to the operating system that runs on the nodes of the Paragon™ supercomputer.

Organization

This manual contains a "manual page" for each command supported by IPD. The manual pages are presented alphabetically. Each manual page provides the following information:

- Command syntax, including all arguments.
- Descriptions of all command arguments.
- A description of what the command does.

Appendix A shows methods used while debugging applications that are programed in the host-node model.

Constants

A number of the IPD commands involve specifying an address or value. To specify an address or value in a number base other than decimal, format the number using the rules of the source language you are working with. For example, octal numbers in Fortran sources have a "0" character (zero) suffix and hex numbers have an "X" suffix. Using C or C++ sources, octal digits have a "0" *prefix* and hex numbers have a "0x" prefix. Assembly language programs may be different—refer to the reference manual for your assembler for information.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, C++ class names, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

`Bold-Italic-Monospace`
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>     <s>     <Ctrl-Alt-Del>`

[ ] (Brackets) Surround optional items.

... (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

Applicable Documents

For more information, refer to the *Paragon™ System Technical Documentation Guide.*

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our products. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056
(44) 793 431062
(44) 793 480874
(44) 793 495108

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

If you have comments about our manuals, please fill out and mail the enclosed Comment Card. You can also send your comments electronically to the following address:

techpubs@ssd.intel.com
(Internet)

Table of Contents

Appendix A
Using IPD With Host/Node Models

ALIAS ALIAS

Display or set aliases.

Syntax

alias [*alias_name* [*command_string*]]

Arguments

alias_name A string (the first character must be a letter) that you choose to represent a command.

command_string The IPD command string that the *alias_name* represents. All of the text following the *alias_name* to the end of the **alias** command line, including any spaces, are part of the *command_string*. To include the pound sign ("#"), semicolons, a non-substituting dollar sign or a blackslash, precede them with a backslash character ("\").

Description

An alias is a character string of your choice that you define to use in place of an IPD command string. Usually, aliases are abbreviations, chosen to save keystrokes. Input on a command line is matched with the list of aliases before it is compared with the IPD command list. A recursive alias definition (an alias that uses the same alias in its definition) is flagged as an error when you use the alias.

Alias arguments that begin with a dollar sign ("$") are substituted for the value of the variable they reference. when the alias is defined. To delay the substitution until the alias is used, escape the dollar sign by preceding it with a backslash character ("\$").

Entering the **alias** command with no arguments lists the current IPD aliases. When you issue the command with the *alias_name* argument alone, the command displays the definition of that *alias_name*. To define a new alias or redefine an existing alias, specify the *alias_name* followed by the *command_string* that defines it.

Use the **unalias** command to delete an alias. The **unalias** and **unset** commands are the only commands that cannot be given an alias.

ALIAS *(cont.)*

ALIAS *(cont.)*

Examples

1. Define an alias for the **step** command:

```
ipd > alias s step
```

2. Define an alias for the commonly used command pair **continue;wait**:

```
ipd > alias cw continue\;wait
```

3. Define an alias for a command and one of its switches:

```
ipd > alias x exec -echo
```

4. Display the current aliases:

```
ipd > alias
  Alias     Command String
  ======    ==============
  x         exec -echo
  cw        continue ; wait
  s         step
```

See Also

unalias, set, unset

ASSIGN ASSIGN

Assign a value to a program expression, address, or register.

Syntax

Assign a value to an expression containing variables in the current scope of context:
assign [*context*] *expression* [*,count*] = *expression*

Assign a value to an expression containing global or static C variables:
assign [*context*] *file*{} *expression* [*,count*] = *expression*

Assign a value to an expression containing a local procedure variable:
assign [*context*] [*file*{}]*procedure*() *expression* [*,count*] = *expression*

Assign a value to an expression containing variables local to a block in C or C++:
assign [*context*] [*file*{}] #*line expression* [*,count*] = *expression*

Assign a value to a program address:
assign [*context*] [-*size_switch*] *address*[:*address*|*,count*] = *expression*

Assign a value to a register:
assign [*context*] [-*size_switch*] *register_switch* = *expression*

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

ASSIGN *(cont.)* ASSIGN *(cont.)*

file

The optional *file* argument is the name of the source module in which the variable resides. To refer to a file other than the location of the current execution point, prefix the variable name with the name of the file where it resides. When you refer to a procedure, you may omit the file name, unless there are duplicate procedure names, because IPD can find the source file from the symbol table information. The *file* argument must end with braces ({}).

procedure

The optional *procedure* argument is the name of the procedure in which the variable resides. You need to specify the procedure or line number when the execution point is not in the same procedure as the variable. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class::

The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::...*.

type

Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

line

A line number from which the variable that you are specifying is accessible. You only need to specify a line number if the variable that you are interested in is hidden by another variable with the same name in the current scope. Specifying any line number from which the variable is accessible allows IPD to find the variable.

ASSIGN *(cont.)* ASSIGN *(cont.)*

variable

The required *variable* argument is the symbolic name of the variable to which you want to assign a value. Alternatively, any expression that can appear on the left side of an assignment may be used in place of a simple variable name. If you specify an array name without a subscript, and IPD can determine the size of the array, each element in the array is assigned *value*.

For C, C++, or Fortran programs, IPD follows the scoping rules of the language in use. For assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information and IPD will use C scoping rules. IPD looks for variables in the following places, in order:

- In the current code block.
- In the current procedure.
- For C++ applications, IPD searches for class members next.
- In the static variables local to the current file.
- In the global program variables.

To specify variables not in the current scope, prefix the variable name with the *file{}*, *procedure()* and/or *#line* qualifiers. C++ class member variables may also be prefaced with the class name, as follows:

```
[[class]::[class::]...]variable
```

Use language-specific syntax to specify a variable. For example, in Fortran you would specify an element of a two-dimensional array as **a(1,1)**; in C or C++, it would be **a[1][1]**.

count

The optional *count* argument is a positive integer that specifies the range of an array variable or address. Designate the beginning array element or address followed by a comma and the *count*; for example, **x(10),10**, or **0x208,8**. This allows you to assign the same value to multiple contiguous elements or addresses.

ASSIGN *(cont.)*

ASSIGN *(cont.)*

size_switch

The *size_switch* switch is an option you can use when you assign a *value* to an address. It specifies how many bytes (1, 2, 4, or 8) are to be assigned to the given address. The *size_switch* command-line switch may be one of the following:

-byte	1 byte
-short	2 bytes
-long	4 bytes
-double	8 bytes

If no *size_switch* is specified when assigning to an address, 4 bytes will be written to that location.

address

The *address* argument is a valid memory address to which you want to assign a value. You can specify a range of addresses with beginning and ending addresses (for example **0x208:0x21b**) or with a starting address and the number of bytes in the range (for example **0x208,20**).

register_switch

The *register_switch* switch argument assigns a value to a register or a floating-point register pair. The value must be numeric. The default size for single-word registers is **-long**. The default may be overridden with the **-double** switch-size switch argument to assign to a floating-point register pair. Similarly, the default size is **-double** for double-word registers (**-KI**, **-KR**, **-T**), but can be overridden with **-long**. Other size specifications (**-byte** and **-short**) are not allowed.

Always specify the even-numbered register of a floating-point register pair. The register switches are **-r0, -r1, -sp, -fp,** and **-r4** through **-r31** for the integer registers, and **-f0** through **-f31** for the floating-point registers. You may also assign to the dual-operation floating-point registers, **-KI**, **-KR** and **-T**, and to the control registers, **-psr, -epsr, -fir, -fsr,** and **-db.** (IPD reports a warning if you try to change the supervisor bits of the control registers.)

expression

The expression representing the value that you want to assign. IPD evaluates expressions containing the following constructs:

constants — All constant types except for Fortran Hollerith constants.

variables — All basic and derived types, except for Fortran unions; C/C++ bit fields; and register variables.

ASSIGN *(cont.)*

ASSIGN *(cont.)*

operators

All Fortran operators, except for function calls and the assignment operator (=). Fortran intrinsic functions are not supported, except for the **loc()** function. All C/C++ operators except the assignment operators (=, *=, %=, +=, /=, -=, <<=, >>=, &=, |=, ^=), function calls, comma(,) and type casts. Overloaded C++ operators, *delete* operators, and *new* operators are not supported.

Values are converted, using C conversion rules, to the type of the variable being assigned. In C or C++ sources, you must enclose a character in single quotes ('*character*') and a string value in double quotes ("*string*"). Fortran strings may be enclosed either in single or double quotes.

Description

The **assign** command changes the value of a variable for the current run. If you re-run the program with the **run** or **rerun** commands, the values of all variables are reset to their original values.

When specifying a variable, use the same language syntax convention as that of the source language. For example, to specify a Fortran array element, you would use **names(1)**; for a C or C++ array element, **names[1]**.

To specify a range of addresses, you can use either the *address***:***address* form or the *address***,***count* form, not both.

You cannot assign values to a structure or union as a whole; you must specify the individual members of a structure or union one at a time.

The **assign** command cannot be used while examining core files.

ASSIGN *(cont.)* ASSIGN *(cont.)*

Examples

1. Assign a new value to the variable *nbrnodes* in the current scope, using a context different from the default:

```
(all:0) > assign (3:0) nbrnodes=3
(all:0) > disp nbrnodes

 ** gauss.f{}shadow()#18 nbrnodes **
  ***** (0..2:0) *****
  nbrnodes = 0
  ***** (3:0) *****
  nbrnodes = 3
```

2. Assign a new value of the expression "node+8" to the variable *iam* in the procedure **shadow()**, using the current context:

```
(3:0) > assign shadow()iam = node+8
(3:0) > display shadow()iam

 ** gauss.f{}shadow()#26 iam **
  ***** (3:0) *****
  iam = 11
```

3. Assign a new value to register 16:

```
(3:0) > assign -r16=2
(3:0) > display -r16

  ***** (3:0) *****
 r16     0x00000002               (2)
```

4. Assign a new value to the variable *i* in the C++ member function **queen::first()**.

```
(3:0) > assign queen::first()i=2
(3:0) > display queen::first()i

 ** queen.C{}queen::first()i **
  ***** (3:0) *****
i = 2
```

ASSIGN *(cont.)* ASSIGN *(cont.)*

5. Assign a new value to the variable *r* in the C++ member function **queen::row(int)**. Note that **row()** may be overloaded.

```
(3:0) > assign queen::row(int)r=8
(3:0) > display queen::row(int)r

 ** queen.C{}queen::row(int)r **
  ***** (3:0) *****
r = 8
```

6. Assign a new value to the variable *col* in the C++ class **queen**. Note that there may be other variables named *col* (local or global) that are visible within the current scope.

```
(3:0) > assign queen::col=8
(3:0) > display queen::col

 ** queen.C{}queen::col **
  ***** (3:0) *****
col = 8
```

7. Assign a new value to the static member variable *board* in the C++ class **queen**.

```
(3:0) > assign queen::board=2
(3:0) > display queen::board

 ** queen.C{}queen::board **
  ***** (3:0) *****
board = 2
```

See Also

display

BREAK BREAK

Set a breakpoint or display current breakpoints.

Syntax

Display breakpoint information:
break [*context*] [**-full**]

Set code breakpoint at procedure entry:
break [*context*] [*file*{}] *procedure*() [**-after** *count*]

Set code breakpoint at source line number:
break [*context*] [*file*{}] [*procedure*()] *#line* [**-after** *count*]

Set code breakpoint at instruction address:
break [*context*] *address* [**-after** *count*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-full Displays breakpoint information in a long or "full" format with more room for class, file and procedure names.

file The *file* argument name of the source module in which the procedure or line resides. To refer to a file that is not where the current execution point is located, prefix the line number with *file*. When you refer to a procedure, you may omit the file name unless there are duplicate procedure names. The *file* argument must end in braces ({}).

BREAK *(cont.)* BREAK *(cont.)*

procedure

The optional *procedure* argument is the name of the procedure at which you want to set the breakpoint, or the procedure in which the line you are specifying resides. If you set a breakpoint at a procedure name, execution is halted just before the first executable line in the procedure, or at the entry point if the first executable line doesn't have a number. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class::

The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

type

Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

-after *count*

The *count* argument is a positive integer indicating the number of times this breakpoint is encountered before execution is halted. The default count is 1. For example, if you have a Fortran loop defined by the following

```
DO 10  I = 1,100
```

and you want IPD to break on every fifth iteration, set the breakpoint on a line in the body of the loop and include the **-after** 5 switch argument.

#line

The *line* argument is the source line number at which you want to set the breakpoint. The line number must be preceded with a pound sign (#). In general, the statement must be executable. For example, you may not set a breakpoint on a Fortran **FORMAT** statement, a comment, or an empty line. The process breaks just before executing the specified statement. To qualify the line number, use the *file*{} and/or *procedure*() qualifiers.

BREAK *(cont.)* BREAK *(cont.)*

address The *address* argument specifies the address at which to set a breakpoint, which must be an instruction address and not a data address. (Use the **watch** command to set a watchpoint on a data address.) The process breaks just before executing the instruction.

Description

The **break** command sets a breakpoint at a specific location in the application that's being debugged. When **break** is used without arguments, it displays the current breakpoints that are set.

When you define a breakpoint, it takes on either the context that you assign it, or the default context. A breakpoint's context denotes the nodes and processes to which it applies. When IPD displays breakpoints, only those breakpoints in the current context are listed.

Entering the **break** command with no arguments displays all breakpoints in the current context. You may also use the **break** command with the *context* argument to display all breakpoints in the specified context. Following is an example of the **break** command display:

```
(all:0)

Bp #  File name  Procedure  Breakpoint Condition        Bp context
===== ========== ========== ======================      ==========
   1  gauss.f    shadow     Line 150                    (all:0)
```

In the preceding display, the first line shows the current context for the **break** command. The labeled columns denote the following:

Bp # The number of each breakpoint. The breakpoint number is used as an argument to the **remove** command.

File name The name of the source file associated with the breakpoint.

Procedure The name of the procedure where the code or variable is located. For global or static variables the "Procedure" field is set to "<global>" or "<static>".

Breakpoint Condition
The condition under which the breakpoint will occur. The **after** clause is not displayed unless the *count* is greater than 1.

BREAK *(cont.)*

BREAK *(cont.)*

Bp context The breakpoint context. If the text overflows the "File name", "Procedure" and "Breakpoint Condition" columns, the right-most characters of the text are truncated. However, if the context overflows the "Bp context" field, the display for the breakpoint is continued on the next line. This is denoted by blanks in all fields except the "Bp context" field, which contains the continued breakpoint context.

In some cases, the file and procedure names may be long enough that truncating them is not a good option. In that case, you may use the **-full** switch to force IPD to use an expanded format for the break display. The expanded format includes separate lines for the file name, procedure name, and breakpoint condition:

```
(all:0)

      File name
      Procedure
Bp #  Breakpoint Condition                          Bp context
====  ======================================        ==========
   1  gauss.C                                       (all:0)
      interval::area(void)
      line 150
```

Breakpoints and tracepoints are not allowed at the same location simultaneously. IPD issues an error message if this is attempted.

If a single C statement consists of multiple source lines, set the breakpoint at the ending line. If a single C++ statement consists of multiple source lines, set the breakpoint at the starting line. For a multiple-line Fortran statement, set the breakpoint on the first line.

When you set a breakpoint on a function, as in this example:

```
   break my_function()
```

the breakpoint is set on the first line of the function, if the function was compiled with symbols. If it was not compiled with symbols, or line number information has been stripped, the breakpoint is set on the function's entry point. As a result, if you set a breakpoint on a function, and then attempt to set a breakpoint on the first executable line of the same function, you will get a "breakpoint already exists" error.

This command returns an error if it is used while examining core files.

BREAK *(cont.)*

BREAK *(cont.)*

Examples

1. Set a breakpoint at the procedure **shadow()** in the current source file for node 0, process type 0 only:

```
(0:0) > b shadow()
```

2. Set a breakpoint at line number 175 in the file *gauss.f*. Set the breakpoint so that the break occurs at the beginning of the tenth execution of the line 175 for process type 0 on nodes 1, 2, and 3:

```
(all:0) > break (1..3:0) gauss.f{}#175 -after 10
```

3. Set a breakpoint at line number 180 in the source file *gauss.f*:

```
(all:0) > break gauss.f{}#180
```

4. Set a breakpoint at the C++ member function **pivot(int*)** for the class **row**. Note that there may be other functions named **pivot()**:

```
(all:0) > break row::pivot(int*)
```

5. Set a breakpoint at the C++ operator function ++ for class **row**:

```
(all:0) > break row::operator++()
```

6. Display the current breakpoints. The **break** command displays those breakpoints that have a process in the current context. The display context is shown on the line before the table and the context of the breakpoint is shown in the rightmost column of the display:

```
(0:0) > break (all:0)
(all:0)
Bp #  File name   Procedure  Breakpoint Condition      Bp context
====  =========   =========  ====================      ==========
   1  gauss.f      shadow     Call shadow                (0:0)
   3  gauss.f      shadow     Line 175 after 10          (1..3:0)
   4  gauss.f      shadow     Line 180                   (all:0)
```

See Also

trace, watch, step

COMMSHOW COMMSHOW

Display the handles (names) for MPI communicators assigned by the debugger.

Syntax

List all handles for communicators:
commshow

Display the handle for a communicator specified by a variable or expression in the current scope of context:
commshow [*context*] *expression*

Display the handle for a communicator specified by an expression containing global or static C variables:
commshow [*context*] *file*{} *expression*

Display the handle for a communicator specified by an expression containing local procedure variables:
commshow [*context*] [*file*{}] *procedure*() *expression*

Display the handle for a communicator specified by an expression containing variables local to a block in C or C++:
commshow [*context*] [*file*{}] *#line expression*

Display the handle for a communicator specified at a memory location:
commshow [*context*] *address*

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

COMMSHOW *(cont.)*   COMMSHOW *(cont.)*

file{}
To specify a communicator using an expression containing global or static C or C++ variables in a source file other than the source file of the current context's execution point, you must qualify the variables with the *file*{} prefix.

procedure()
To specify a communicator using an expression containing local variables in a procedure other than the procedure of the current context's execution point, you must qualify the variables with the *procedure*() prefix. The *file*{} prefix can be omitted since IPD can find the source file from the symbol table information. However, if you have procedures with the same name in two different source modules then the name must be fully qualified with a *file*{} prefix.

For C++, procedure names may include operator functions such as **operator+()**, **operator new()**, or **operator int** *(); the operator name must include the "operator" keyword. Also for C++, you may preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The full syntax is as follows:

[[*class*]**::**[*class***::**]...]*procedure*([*type*[, *type*]...])

*class***::**
The name of the C++ class in which a procedure is a member function. Use **::** without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1***::***class2***::**....

type
Any legal C++ type specification such as *int*, *float* *, *char (*)()*, etc. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only specify enough arguments to uniquely identify the desired procedure. An error is reported when the procedure name is ambiguous.

#line
A line number from which the variable that you are specifying is accessible. You only need to specify a line number if the variable that you are interested in is hidden by another variable of the same name in the current scope. In that case, specifying any line number from which the desired variable is accessible allows IPD to find the variable.

address
Display the handle of the communicator pointed to by the contents of the memory location specified by the *address* argument.

expression
The expression representing the location (address) of the communicator whose handle you want to display.

COMMSHOW *(cont.)*

COMMSHOW *(cont.)*

When displaying expressions containing variables, IPD uses appropriate scoping rules for the language in which you are programming (C, C++ or Fortran). For Assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information and IPD will use C scoping rules.

To specify expressions not in the current scope, prefix the expression name with the *file*{}, *procedure*() and/or *#line* qualifiers. For C++ programs, class member variables may also be prefaced with the class name(s) as follows:

[[*class*]**::**[*class***::**]...]*variable*.

IPD also supports expression evaluation and can handle the following language constructs:

constants All constant types except Fortran, octal, hexadecimal, and Hollerith constants.

variables All basic and derived types except Fortran structures, records and unions; and C/C++ bit fields. Also, register variables are not supported.

operators All Fortran operators except function calls and the assignment operator (=). All C/C++ operators except the assignment operators (=, *=, %=, /=, +=. -=, <<=, >>=, &=, |=, ^=), function calls, comma (,), and type casts.

Examples of valid C/C++ expressions include the following: "a[i+j] - 3 * k", "b[3]", "sizeof(int) * 3 - (foo ? i + j : k / 5)".

Examples of valid Fortran expressions include the following: "a(3,k) - f * 5.0", "iary(7-k*j) / i(12,l,m+2) ** 2".

Description

The **commshow** command determines the handle (name) assigned to a MPI communicator by the debugger. The argument supplied to **commshow** should resolve to a variable of type *MPI_Comm*, which has previously been assigned a communicator. Optionally, an address may be specified that is treated as a pointer to an object of type *MPI_Comm*. An error occurs if the specified variable is of the wrong type or if its contents do not map to a valid communicator. If successful, the name of the communicator displayed by **commshow** is suitable for use in a context command or argument.

COMMSHOW *(cont.)* COMMSHOW *(cont.)*

Without any arguments, **commshow** displays the known list of communicator handles that may be specified in a context command or argument. The size of the communicator's group and the rank (relative to *MPI_COMM_WORLD*) is displayed along with the handle. The display for intercommunicators consists of the handle for the intercommunicator followed by the handle for each of the intracommunicators that comprise it.

When MPI is initialized, several communicators are created automatically. The first is *MPI_COMM_WORLD* and it is given the handle *COMMWORLD*. The others are *MPI_COMM_SELF*. Each process within *MPI_COMM_WORLD* creates its own self communicator; They are given the handles *COMMSELF0, COMMSELF1*, ... *COMMSELFn-1,* where n is equal to the number of processes in *MPI_COMM_WORLD*.

Each time an intracommunicator is created using MPI functions like *MPI_Comm_create*(), *MPI_Comm_dup*(), or *MPI_Comm_split*() the debugger attaches a name to it of the form *COMM1*, *COMM2*, ... in the order that the communicators are created by the application. It is possible for a communicator to be assigned a different handle from run to run whenever communicators are created on different processes that do not include all other processes.

Intercommunicators created via *MPI_Intercomm_create*() are assigned the names *ICOMM1*, *ICOMM2*, ..., where the numbering is kept separate from the intracommunicators.

An intercommunicator name may be used in a context command or argument only in conjunction with a *ranklist* specification of **all**. This restriction is caused by the fact that an intercommunicator and rank value do not uniquely identify a process when expressed outside of message passing function calls.

Execution of the call *MPI_Comm_free*() results in that communicator's handle being deleted. If at anytime a communicator is deleted whose handle is used in an IPD context, the *COMMWORLD* handle replaces it.

The **commshow** command changes slightly when used with core files:

- **commshow** does not display communicators unless full core dumps exist, because processes that are part of an MPI application can only be identified if this is the case.

- Only *COMMWORLD* and *COMMSELF* communicators appear initially. Then, each time you use **commshow** with a new expression for a valid communicator, the debugger creates a handle for it and adds it to the list of communicators that can be used in context specifications.

- The intracommunicator handles that are displayed for an intercommunicator do NOT show the handles of the original intracommunicators that created it, as they do for runtime debugging.

COMMSHOW *(cont.)* COMMSHOW *(cont.)*

Examples

1. List the communicators for an MPI application that has executed the creation of several communicators:

```
(COMMWORLD:all) > commshow
Intracommunicators:
Name        Size    Rank (in COMMWORLD)
=======     =====   ===================
COMMWORLD      4    0..3
COMM1          4    0..3
COMM2          1    2
COMM3          2    0,3
COMM4          1    1

COMMSELF[0..3]

Intercommunicators:
Name        Intracommunicator Pair
=======     ======================
ICOMM1      COMM3,COMM4
ICOMM2      COMM2,COMM4
```

2. Display the debugger's name for the communicator in variable *myFirstComm*:

```
((COMMWORLD:all) > commshow myFirstComm
  ***** (COMMWORLD:0,1,3) *****

 ** inter2.c{}main()#35 myFirstComm **
myFirstComm = ICOMM1

  ***** (COMMWORLD:2) *****

 ** inter2.c{}main()#35 myFirstComm **
myFirstComm = ICOMM2
```

See Also

context, coreload, msgstyle

CONTEXT CONTEXT

Set the debug context, defining the default set of processes and nodes to which debug commands apply.

Syntax

Set the debug context to compute partition processes:
context ({**all** | **nodes** | *nodelist*} : {**all** | *ptypelist*})

Set the debug context to the host process (host/node model):
context (host)

Set the debug context to service partition processes (host/node model):
context (host : {**all** | *ptypelist* })

Set the debug context using rank values as process identifiers (MPI applications):
context (*communicator* : {**all** | *ranklist*})

Display the context in which the application was loaded:
context [-pid]

Arguments

all

When used on the left side of a colon, the **all** argument specifies all compute nodes. On the right side of a colon, the **all** argument specifies all *ptypes* under debug on the specified nodes. When used with an MPI communicator **all** specifies all ranks within the communicator's group.

nodes

The **nodes** argument is an alias for **all** (all compute nodes).

nodelist

The *nodelist* argument specifies the set of nodes to be used with debug commands. A single value indicates a single node. You may specify a range of nodes with the syntax *node1..node2*, where node2 is greater than node1. Specify a list of nodes by separating node numbers with commas, using the syntax *node*, *node*, *node,...node*. The node specification may include both a range of nodes and a list of nodes. Rather than a list of nodes, you may use the special value **all**, which specifies all compute nodes where loaded processes reside.

CONTEXT *(cont.)*

CONTEXT *(cont.)*

ptypelist The *ptypelist* argument is a single value that indicates a process type. You may specify a range of process types with the syntax *ptype1..ptype2*, where *ptype2* is greater than *ptype1*. Specify a list of process types by separating process type numbers with commas, using the syntax *ptype, ptype, ..., ptype*. The *ptypelist* may include both ranges and lists of process types. Rather than a list of process types, you may use the special value **all**, which specifies the process types of all loaded processes under debug on the specified compute nodes.

host When used by itself, specifies the host (controlling) process in applications using a host/node model of computation. When used with a *ptype* specification, refers to all processes in the service partition.

communicator The debugger-assigned handle for an MPI communicator. Use the **commshow** command to see a list of valid handles.

ranklist A rank is an identifier for a process within an MPI communicator's group of processes. A single value indicates a single process. You can specify a range of ranks with the syntax *rank1..rank2*, where *rank2* > *rank1*. Specify a list of ranks by separating rank numbers with commas, using the syntax *rank, rank, rank, ..., rank*. The *ranklist* may include both a range of ranks and a list of ranks. Alternatively, the special value **all** may be used to specify all ranks in the communicator's group.

-pid The **-pid** switch argument indicates that the ID for each process should be included in the display.

Description

The default context is first set with the **load** or **coreload** command and is displayed as part of the IPD prompt. For programs compiled with the **-nx** switch, the **load** or **coreload** command picks one process type created and sets the default context to be all processes sharing that *ptype*. For all other programs, the default context is set to **host**.

The **coreload** command forms a default context that includes all of the processes whose core files are being loaded. In this case, the keyword **all** is used relative to the partition in which the faulting application was running, not the partition in which the core file analysis is being done. In addition, when performing core file analysis, the keyword **all** is allowed even if all of the processes in the faulting application are not loaded for analysis. A message is printed for those processes without a core file. This is allowed to provide a shorthand method of referencing a disjoint list of processes during postmortem debug.

You may change the default context with the **context** command. When you need to override the default context for a given command, specify the context as part of the command syntax. This override is valid only for that command.

CONTEXT *(cont.)* CONTEXT *(cont.)*

A communicator and rank cannot be used to specify a context until after the *MPI_Init()* routine within an application is executed. Once *MPI_Init()* is executed, context specifications are assumed to be of the form (*communicator:ranklist*) until the *MPI_Finalize()* routine is executed. Use of the keyword **all** on the right side of the : specifies all ranks in the communicator's group (0 to n-1, where n is the number of processes in the group). If a *node/ptype* pair is used to specify a context while debugging an application initialized for MPI, the node list is interpreted as a list of ranks relative to MPI_COMM_WORLD and then converted to its corresponding rank value for whichever communicator is in the default context. If the process is not a part of the default context communicator, an error is reported. To force the use of a context as the NX style (*node:ptype*), use the command **msgstyle**. This same command sets the context style back to MPI. When the mode is set back to MPI, the context reverts back to the communicator that was in the default context when the switch to the NX mode occurred. If the node list was changed while in the NX mode, so that all of them do not exist in that communicator, the COMMWORLD communicator is used in the default context instead. In either case, the list of nodes in the default context at the time the mode is switched remains the same. Only their rank may change, depending upon which communicator they are being associated with.

A list of communicator names is obtained from the command **commshow**. The name COMMWORLD is always present and represents the predefined communicator MPI_COMM_WORLD, which denotes all available processors in the application partition.

Without arguments, the **context** command displays all loaded processes and their executables.

If the **-pid** switch is specified, the process ID associated with each *node/ptype* pair is included in the display. For host/node applications (not compiled with **-nx**), the host process information is included in the display.

When debugging a host/node application, a combined host/node context can be created by specifying the *nodelist* explicitly (not using "**all**"). For example, if an application is loaded on four nodes, the host node ID is 4, but the "**all**" includes only nodes 0...3. Specifying 0...4 as the node list includes the host in the context. Be careful when including the host node in the node list, because it does not share a common *ptype* with the other nodes and many commands may return errors.

The **context** command can only refer to existing processes under debug.

The **coreload** command forms a default context that includes all of the processes from core files that are being loaded. In this case, the keyword **all** refers to the partition in which the faulting application was running, not the partition where IPD is running.

CONTEXT *(cont.)* CONTEXT *(cont.)*

Examples

1. Set the context to include process type 0 on all compute nodes:

```
(host) > context (all:0)
(all:0) >
```

2. Set the context to the controlling (host) process:

```
(all:0) > context (host)
(host) >
```

3. Set the context to include all processes in the MPI_COMM_WORLD group:

```
(host) > context (COMMWORLD:all)
(COMMWORLD:all) >
```

4. Change the context to include only the first 4 processes of the communicator:

```
(COMMWORLD:all) > context (COMM1:0..3)
(COMM1:0..3) >
```

5. Display the context for a host/node application:

```
(0:0)> context

Nodes          Ptype       Program
==========     =======     =======
(0..3)           0         gauss.nodes
4                0         gauss.host
```

6. Display the host/node application context with process IDs:

```
(all:0) > context -pid

 Pid            (Node:Ptype)      Program
 ==========     ============      =======
 35678              (0:0)         gauss
 35679              (1:0)         gauss
 35680              (2:0)         gauss
 35681              (3:0)         gauss
 45635              (4:0)         gauss.host
```

CONTEXT *(cont.)* CONTEXT *(cont.)*

See Also

commshow, **msgstyle**

CONTINUE CONTINUE

Continue execution of processes stopped by command or breakpoint in the current context.

Syntax

continue [*context*] [**-nosignal**]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-nosignal Deliver no signal upon execution. By default, IPD delivers any pending signal.

Description

The **continue** command resumes execution of stopped processes. The processes may be stopped as the result of a **stop** command or by the action of breakpoints or watchpoints. It may also be used to start processes that have just been loaded.

After the processes have been continued, IPD returns control to you by issuing the next IPD prompt. To cause IPD to delay returning control to you until all processes in the context stop (terminate or hit breakpoints), use the **wait** command.

If a process is currently stopped at a signal, IPD delivers that signal by default when the process is restarted. The **-nosignal** switch can be used to change that default action.

This command may not be used while examining a core file.

CONTINUE *(cont.)*

CONTINUE *(cont.)*

Examples

1. Continue executing a single process with type 0 on node 1 when the default context is "(all:0)".

```
(all:0) > continue (1:0)
(all:0) >
```

2. Continue all processes in the default context.

```
(all:0) > continue
(all:0) >
```

3. Continue all processes in the default context, delivering no signals.

```
(all:0) > continue -nosignal
(all:0) >
```

4. Continue all processes and wait for them to stop before returning a prompt.

```
(all:0) > continue; wait
```

See Also

run, wait, signal, stop, rerun

CORELOAD CORELOAD

Load core files for examination.

Syntax

coreload [-all | -fault | -first | -nonfault | *context***]** [*core_name*] [**-pn** *partition*] [**-sz** *size*]

Arguments

-all

The **-all** switch selects all core files.

-fault

The**-fault** switch selects core files belonging to faulting processes only. (This is the default.)

-first

The **-first** switch selects the core file of the first process that faulted. An internal time stamp is used to determine this, not the time stamp on the core files themselves.

-nonfault

The **-nonfault** switch selects core files belonging to all non-faulting processes.

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

core_name

The *core_name* argument specifies the name of a core file or directory. It is used to override the default name and/or location named *core*. If a file name is specified, that file is loaded as a core file. If a directory is specified, a file or directory named *core* within that directory is looked for. If that is not found, the directory specified is assumed to be a renamed core directory and is treated as such.

CORELOAD *(cont.)* CORELOAD *(cont.)*

-pn *partition* The **-pn** *partition* switch argument specifies the name of the partition to use for core file examination. The *partition* argument is a string representing the name of a previously created partition in *.compute*. The default partition is used if this is not specified.

-sz *size* Restricts the number of nodes to use for core file examination to the value of *size*. The *size* argument is a positive integer that is less than or equal to the maximum number of nodes in the partition. All nodes in the partition may be used if this is not specified.

Description

The **coreload** command loads one or more core files for examination. By default, a file or directory named *core* is looked for in the directory specified in the *CORE_PATH* environment variable, if it is set, otherwise it is assumed to be in the current directory. If a core directory is found, only the processes that terminated due to a fault are loaded. The switches **-all**, **-nonfault**, and **-first** change which group of processes within a core directory are loaded for examination. The *core_name* argument overrides the default name and location of *core*.

After the core files are loaded, the **process** command is automatically executed to display summary information about all of the processes that are loaded. The default context is set to include the nodes and ptype of the processes that are loaded.

Upon completion of the loading of the core files, the **process** command automatically displays summary information about all of the processes that are loaded. The default context is set in the same fashion as for the **load** command (a single ptype and all nodes on which it is found).

A FULL core dump (not TRACE) is required to determine whether the application that dumped the core was using MPI. If MPI was initialized at the time of the fault, the context takes the form of a communicator and a list of ranks. The communicator will be COMMWORLD and the ranks listed will be relative to it. Refer to the **commshow** command for details on the communicator handles available after a **coreload.**

The only requirements for the partition chosen for performing core file analysis is that it reside in the compute partition and contain at least one node. It does not have to be equivalent in size to that used by the application which faulted. Larger partitions (i.e greater than 1) allow parallel collection of data to occur and are therefore recommended when examining many core files.

The partition chosen for performing core file analysis must reside in the compute partition and must contain at least one node. It does not have to be equivalent in size to that used by the application that faulted. Larger partitions (greater than one) allow parallel collection of data to occur and are therefore recommended when examining many core files.

CORELOAD *(cont.)* CORELOAD *(cont.)*

The **-pn** and **-sz** switch commands allow control over which partition and how many nodes of a partition are used for core file analysis. The default partition (determined by the environment variable *NX_DFLT_PART*) is assumed if the **-pn** switch command is not used. All nodes in a partition may be used if the **-sz** switch command is not specified. The actual number of nodes used in a partition will not exceed the number of core files available in a core directory.

After using the **coreload** command, all references to node numbers and ptypes should be taken in the context of the application that dumped the core file(s) being examined.

Once the **coreload** command is used, commands are limited to those that are not related to executing code. The **help** command shows this reduced list when used after a **coreload** command. To get out of core file mode either use a **load** command or empty the list of core files being examined with this command:

```
kill (all:all)
```

Note that reducing the context via the **context** command is not equivalent to "killing" a core file. Use the **kill** command to permanently remove a process's core file from analysis. Use the **context** command to temporarily exclude a process from consideration.

Multiple uses of the **coreload** command are cumulative if they load core files from the same core directory. The core files specified are added to the list of those available for examination and duplicates are silently ignored. An asterisk (*) in the **process** display that completes each core load indicates the new processes that have been loaded.

NOTE

The default context (as displayed in the prompt) is not automatically updated to include the new processes. Use the **context** command to add them to the default context.

Cumulative uses of the **coreload** command assume that the same partition name and size should be used as specified in the initial **coreload** command, unless a new partition name or size is explicitly specified via the **-pn** and/or **-sz** switch arguments. Changing the debug partition in this fashion constitutes a new initial core load rather than a cumulative core load.

A parallel core directory is identified by the existence of the *allocinfo* file. If this file is missing, only a single core file can be loaded for examination and it is assumed to belong to a non-NX application.

A warning is displayed for core files with a corresponding executable that is newer than the core file's internal time stamp, or if the executable file cannot be found.

CORELOAD *(cont.)* CORELOAD *(cont.)*

Use the **status**, **process**, and **context** commands to get more information about the application being examined after a **coreload** command completes.

Examples

1. Load core files for all faulting processes:

```
ipd > coreload
 *** reading symbol table for
/home/karla/tests/fault/segfault... 100%
 *** scanning core files...
 *** coreload complete
 Context  State    Reason   Location    Procedure
 ======= =====     ======   ========    =========
*(all:0) Signaled  SIGSEGV  Line 9      sub1()
(all:0) >
```

2. Load only one core file, the first that took a fault. Then add the core file for process (1:0) for examination. Note that the default context is not changed by the second **coreload** command. The process display indicates that the core file for process (1:0) has been loaded and can be added to the default context if desired. The asterisk (*) indicates that this process is newly loaded:

```
ipd > core -first
 *** reading symbol table for
/home/karla/tests/fault/segfault... 100%
 *** scanning core files...
 *** coreload complete
 Context  State    Reason   Location    Procedure
 ======= =====     ======   ========    =========
*(0:0)   Signaled  SIGSEGV  Line 9      sub1()
(0:0) > core (1:0)
 *** scanning core files...
 *** coreload complete
 Context  State    Reason   Location    Procedure
 =======  =====    ======   ========    =========
 (0:0)    Signaled SIGSEGV  Line 9      sub1()
*(1:0)    Signaled SIGSEGV  Line 9      sub1()
(0:0) >
```

CORELOAD *(cont.)* CORELOAD *(cont.)*

3. To load a core file or directory that resides some place other than the current working directory, specify that directory. If you started the IPD program from within the directory */home/joe*, but the core directory you want to examine resides in */home/joe/gauss*, enter the following:

```
ipd > core gauss
*** reading symbol table for /home/joe/gauss/gauss.nx... 100%
*** scanning core files ...
*** coreload complete
 Context  State    Reason   Location    Procedure
 =======  =====    ======   ========    =========
*(0:0)    Signaled gauss.f  Line 20     exchange()
(0:0) >
```

4. Load the core file of a UNIX application (compiled without the **-nx** option).

```
ipd > coreload
 *** reading symbol table for /home/karla/apps/myapp...    100%
 *** scanning core files...
 *** coreload complete
 Context  State    Reason   Location    Procedure
 =======  =====    ======   ========    =========
*(host)   Signaled SIGSEGV  Line 230    getval()
(host) >
```

5. Load core files for faulting processes on only one node of the default partition. This results in multiple core files being loaded on a single node for analysis.

```
ipd > coreload -sz 1
 *** reading symbol table for /home/joe/gauss/gausss.nx... 100%
 *** scanning core files...
 *** coreload complete
 Context  State    Reason   Location    Procedure
 =======  =====    ======   ========    =========
*(all:0)  Signaled SIGBUS   Line 20     exchange()
(all:0) >
```

CORELOAD *(cont.)*

CORELOAD *(cont.)*

6. Exit the postmortem debug session and load an application for normal runtime debugging:

```
ipd > coreload -sz 1
 *** reading symbol table for /home/joe/gauss/gausss.nx... 100%
 *** scanning core files...
 *** coreload complete
 Context  State      Reason    Location      Procedure
 =======  =====      ======    ========      =========
*(all:0) Signaled    SIGBUS    Line 20       exchange()
(all:0) > load gauss.nx
 * This command will terminate the core file analysis session.
   Are you sure you want to do this (y/n)? y
 *** reading symbol table for /home/joe/gauss/gauss.nx... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
(all:0) >
```

See Also

load, status

DISASSEMBLE DISASSEMBLE

Display machine code listing of process instructions.

Syntax

Disassemble from current execution point:
disassemble [*context*] [,*count*]

Disassemble starting from an instruction address:
disassemble [*context*] *address* [:*address* | ,*count*]

Disassemble starting from procedure:
disassemble [*context*] [*file*{}] *procedure*() [,*count*]

Disassemble starting from a source line number:
disassemble [*context*] [*file*{}] [*procedure*()] #*line* [: #*line* | ,*count*]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

count

The *count* argument is an integer that indicates the number of assembly instructions to disassemble. If the value of *count* is positive, disassembly starts at the specified line number. If negative, disassembly begins at *count*-1 instructions preceding the specified point and ends at this point. If you do not specify a *count*, the last *count* argument given to the **disassemble** command is used. Upon starting the IPD program, the initial value of *count* is 50. One way to use the *count* argument is to specify a large value and use the IPD **more** utility to browse through the instructions (see the **more** command).

DISASSEMBLE *(cont.)* DISASSEMBLE *(cont.)*

address The *address* argument specifies where to start the disassembly. Specify a range of addresses by including a count following the address, or by specifying the beginning and ending addresses.

file The *file* argument is the name of the source module in which the procedure or line resides. To refer to a file in which the current execution point is not located, specify the file name as a prefix to the line number. When you refer to a procedure, you may omit the file name unless there are duplicate procedure names in different files. The *file* argument must end with braces ({}).

procedure The optional *procedure* argument is the name of the procedure at which you want to start disassembling, or the procedure in which the line you are specifying resides. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class:: The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::...*.

type Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

line The *line* argument is the source line number at which to start disassembly. The line number must be prefixed by a number sign (#) and must exist in the symbol table debug information. Specify a range of lines by specifying beginning and ending line numbers, or by specifying the beginning line number and a count.

DISASSEMBLE *(cont.)* DISASSEMBLE *(cont.)*

Description

The **disassemble** command allows you to display assembly language code. The contents of the program's address space in memory are disassembled, rather than the contents of the executable file. The target processes must be stopped to perform the disassembly. If they are not stopped, an error message is displayed.

If you enter the command without specifying a starting point (using the current execution point), and the processes within the current context are stopped at different locations in the load module, multiple disassembly lists are displayed, one for each process with a unique execution point.

If the specified procedure or address matches a source line number, that line number is displayed before the instructions. If there is no matching line number, the procedure name + address offset is shown, as in the following example:

```
procedure() + 0x25.
```

DISASSEMBLE *(cont.)* DISASSEMBLE *(cont.)*

Examples

1. Assume that the current context is (all:0) in a Fortran program. Disassemble 30 instructions, starting at the procedure **shadow()**.

```
(all:0) > disa shadow(),30
  ***** (all:0) *****
  gauss.f{}shadow() + 0x0
00000b18: ec1f1001  orh        0x1001, r0, r31
00000b1c: e7ff1c00  or         0x1c00, r31, r31
00000b20: 1fe01801  st.l       fp, 0(r31)
00000b24: a3e30000  mov        r31, fp
00000b28: 1fe00805  st.l       r1, 4(r31)
00000b2c: 1c7f87fd  st.l       r16, -4(fp)
00000b30: 1c7f8ff9  st.l       r17, -8(fp)
00000b34: 1c7f97f5  st.l       r18, -12(fp)
00000b38: 1c7f9ff1  st.l       r19, -16(fp)
00000b3c: 1c7fa7ed  st.l       r20, -20(fp)
00000b40: 1c7fafe9  st.l       r21, -24(fp)
00000b44: 1c7fb7e5  st.l       r22, -28(fp)
  gauss.f{}shadow()#165
00000b48: 147cffe9  ld.l       -24(fp), r28
00000b4c: 1470fffd  ld.l       -4(fp), r16
00000b50: 139d0001  ld.l       r0(r28), r29
00000b54: 12110001  ld.l       r0(r16), r17
00000b58: 97be0002  adds       2, r29, r30
00000b5c: 0810f000  ixfr       r30, f16
00000b60: 08128800  ixfr       r17, f18
00000b64: 1c7ff7d9  st.l       r30, -40(fp)
00000b68: 96320001  adds       1, r17, r18
00000b6c: 4a1491a1  fmlow.dd   f18, f16, f20
00000b70: 1c7f97d1  st.l       r18, -48(fp)
00000b74: 1c7f8fe1  st.l       r17, -32(fp)
00000b78: 1c7f8fdd  st.l       r17, -36(fp)
00000b7c: 2c74ffd6  fst.l      f20, -44(fp)
  gauss.f{}shadow()#175
00000b80: 147cfff1  ld.l       -16(fp), r28
00000b84: 139d0001  ld.l       r0(r28), r29
00000b88: 97beffff  adds       -1, r29, r30
00000b8c: 1c7ff7cd  st.l       r30, -52(fp)
(all:0) >
```

See Also

list

DISPLAY DISPLAY

Display the value of the specified variable, expression, memory address, or processor registers.

Syntax

Display the value of expression in current scope of context:
display [*context*] [*-format_switch*] *expression* [*,count*]

Display the value of an expression containing global or static C variables:
display [*context*] [*-format_switch*] *file*{} *expression* [*,count*]

Display the value of an expression containing a local procedure variable:
display [*context*] [*-format_switch*] [*file*{}]*procedure*() *expression* [*,count*]

Display the value of an expression containing variables local to a block in C or C++:
display [*context*] [*-format_switch*] [*file*{}] *#line expression* [*,count*]

Display the value of a memory address:
display [*context*] *address*[*:address* | *,count*]

Display the contents of all processor registers:
display [*context*] **-register**

Display the contents of one processor register:
display [*context*] [*-format_switch*] *-register_name*

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

DISPLAY *(cont.)*

DISPLAY *(cont.)*

format_switch

The *format_switch* overrides the symbol table information that would normally determine how a symbol's value would be printed. For variable and memory address displays, the *format_switch* can be one of the following:

alphanumeric
complex
dcomplex
decimal
double
float
hexadecimal
octal
real (equivalent to the C float type)
string (see Description)
logical

For register displays, the *format_switch* can be one of the following:

decimal
double
float
hexadecimal
real

count

One of the ways to specify a range is to specify the beginning array element followed by a comma and a *count* (for example, **x(10),10**). You must specify the range of an array in ascending order. If you only use the array name without a subscript, all elements in the array are displayed.

file{}

Use the *file* argument to display an expression containing global or static C or C++ variables in a source file other than the source file of the current context's execution point.

procedure()

You must specify the name of the procedure or the line number that contains the variables you want to display if they reside in a procedure other than the one with the current execution point. The *file* argument may be omitted, because IPD can find procedures from the symbol table information; If you have procedures with the same name in two different source modules, the name must include the *file* argument.

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class::

The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

DISPLAY *(cont.)* DISPLAY *(cont.)*

type Any legal C++ type specification, such as *int, float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

#line A line number from which the variable that you are specifying is accessible. You only need to specify a line number if the variable you are interested in is hidden by another variable of the same name in the current scope. Specifying any line number from which the desired variable is accessible allows IPD to find the variable.

address Display the contents of the memory location specified by the *address* argument.

There are two ways to denote a range of memory locations. You can either specify the beginning address and the ending address, separated by a colon (for example, **0x208:0x21b**), or you can specify the beginning address followed by a comma and a count (for example, **0x208,10**).

-register Display all of the processor registers.

-register_name Display a specific processor register. The register names follow the processor naming conventions. See the Description section for a complete list of register names.

expression The expression representing the value that you want to display. IPD evaluates expressions containing the following constructs:

constants All constant types except for Fortran Hollerith constants.

variables All basic and derived types, except for Fortran unions; C/C++ bit fields; and register variables.

operators All Fortran operators, except for function calls and the assignment operator (=). Fortran intrinsic functions are not supported, except for the **loc()** function. All C/C++ operators except the assignment operators (=, *=, %=, +=, /=, -=, <<=, >>=, &=, |=, ^=), function calls, comma(,) and type casts. Also, overloaded C++ operators, the *new* operator, and the *delete* operator are not supported.

DISPLAY *(cont.)* DISPLAY *(cont.)*

Values are converted, using C conversion rules, to the type of the variable being assigned. In C or C++ sources, you must enclose a character in single quotes (‘*character*’) and a string value in double quotes (“*string*”). Strings in Fortran sources may be enclosed in either single or double quotes.

Description

The **display** command examines the current value of an application variable, expression, or data location.

When specifying a variable or expression, use the same language syntax convention as that of the source language. For example, to specify a Fortran element, you would use **names(1)**; for a C or C++ element, **names[1]**. For assembly programs, you may use either C or Fortran syntax to display a memory address.

To specify a range of addresses, you can use either the *address*:*address* form or the *address*,*count* form.

You can display all of the elements of an array by specifying its name. You cannot display all of the elements of a structure by specifying its name; you must specify individual elements.

When displaying from a single address, IPD prints 352 bytes by default in a formatted hex dump.

Use the **-string** switch to display a C or C++ character array as a null-terminated string. Otherwise, it is displayed as individual characters. For example, in a C program with a variable declared to be **char name[5]**:

```
(1:0) > display name
name[0] = J
name[1] = o
name[2] = e
name[3] = y
name[4] =\0000
(1:0) > display -string name
name = "Joey"
```

DISPLAY *(cont.)* DISPLAY *(cont.)*

Use the **-register** switch to display all registers. The following switches are recognized for displaying individual registers:

-r0, -r1, -sp, -fp, -r4 .. **-r31**	Integer register set
-f0 .. **-f31**	Floating-point register set
-psr and **-epsr**	Processor status register and extended processor status register
-db	Data breakpoint register
-dirbase	Directory base register
-fsr	Floating-point status register
-fir	Fault instruction register
-KI, **-KR** and **-T**	"Konstant" registers and "temporary register"
-merge	Merge register

The register displays include the hexadecimal representation of the register(s) followed by an interpretation. The default interpretation is decimal for the integer registers and floating-point for the floating-point registers. Use the *format_switch* parameter to change the default interpretation.

For C or Assembly programs, IPD follows the C scoping rules. It first looks for a variable in the current code block, then in the current procedure. Next, it looks for the variable in the static variables local to the current file, and finally in the global program variables. Similarly, for C++ program IPD follows C++ scoping rules, which are similar to C scoping rules, except that class variables are searched prior to static and global variables.

The display of registers and local variables is affected by the setting of the **threads** command. If it is set to "on" a value for each thread in each process will be displayed. Other data values will be the same across all threads and are thus unaffected by the thread setting.

DISPLAY *(cont.)* DISPLAY *(cont.)*

Examples

1. Display 20 elements of the array *a*, starting at *a(1,4)*. To display the entire array, you would simply specify the array name. This listing uses the same column-major indexing as the Fortran program.

```
(all:0) > disp (0:0) gauss()a(1,4),20
 ***** (0:0) *****

 ** gauss.f{}gauss()#32 a(1,4) **
  a(1,4) = 0.0000000000000
  a(2,4) = 3.1250000000000
  a(3,4) = 5.4687500000000
  a(4,4) = 6.6406250000000
  a(5,4) = 7.1289062500000
  a(6,4) = 7.3120117187500
  a(7,4) = 7.3760986328125
  a(8,4) = 7.3974609375000
  a(9,4) = 7.4043273925781
  a(10,4) = 7.4064731597900
  a(11,4) = 7.4071288108826
  a(12,4) = 7.4073255062103
  a(13,4) = 7.4073836207390
  a(14,4) = 7.4074005708098
  a(15,4) = 7.4074054602534
  a(16,4) = 7.4074068572372
  a(17,4) = 7.4074072530493
  a(18,4) = 0.0000000000000
  a(19,4) = 0.0000000000000
  a(20,4) = 0.0000000000000
```

2. Display the variable named *iam* in the default context.

```
(0:0) > display iam
 ***** (0:0) *****

** gauss.f{}gauss()#32 iam **
  (0:0) iam = 4
```

DISPLAY *(cont.)*

DISPLAY *(cont.)*

3. Display floating point register f12.

```
(all:0) > disp -f12

  ***** (all:0) *****
f12       0x3f99999a           ( 1.200000 )
```

4. Display the value of the Fortran expression "iam + nodes(i)".

```
(1:0) > disp iam + nodes(i)
  ***** (1:0) *****

 ** gauss.f{}gauss()#32 iam+nodes(i) **
  iam + nodes(i) = 4
```

5. Display the variable *row* in C++ function **nextMove**, where **nextMove** is an overloaded function name (several functions with the same name but different numbers or types of arguments):

```
(all:0) > disp nextMove(int, struct move*)row
     ***** (all:0) *****

 ** moves.C{}nextMov(int, struct move*)#44 row **
  row = 3
```

6. Display C++ static class member variable *occupied* for the nested class **board::square**:

```
(all:0) > disp board::square::occupied
     ***** (all:0) *****

 ** board.C{}board()#12 board::square::occupied **
  occupied = 1
```

7. Display variable *row* in C++ member function **position** for the class **board::square**:

```
(all:0) > disp board::square::position()row
     ***** (all:0) *****

 ** board.C{}board::square::position()#11 row **
  row = 1
```

DISPLAY *(cont.)* DISPLAY *(cont.)*

8. Display from a memory address.

```
(all:0) > display 0xbffffd8c
  ***** (all:0) *****
0xbffffd8c: 0x00000004 0xbffffe00 0x00010178 0x00000000   *........x.......*
0xbffffd9c: 0x00000000 0x00000000 0x00000000 0x00000000   *................*
0xbffffdac: 0x00001000 0x00000064 0x00000381 0x00000747   *....d.......G...*
0xbffffdbc: 0x000004d4 0x00000001 0xbffffe64 0xbffffe74   *........d...t...*
0xbffffdcc: 0x00001a93 0x00017061 0x00001849 0x00000ef9   *....ap..I.......*
0xbffffddc: 0x00000a00 0xbffffe00 0x00010134 0x00000003   *........4.......*
0xbffffdec: 0xbffffe64 0xbffffe74 0x00000000 0x00000000   *d...t...........*
0xbffffdfc: 0x00000000 0x00000000 0xdeadc0de 0x00000000   *................*
0xbffffe0c: 0x00000000 0x00000000 0x00000000 0x00000000   *................*
0xbffffe1c: 0x00000000 0x00000000 0x00000000 0x00000000   *................*
0xbffffe2c: 0x00000000 0x00000001 0xbffffe64 0xbffffe74   *........d...t...*
0xbffffe3c: 0x00000000 0x00000000 0x00000000 0x00000000   *................*
0xbffffe4c: 0x00000000 0x00000000 0x000100d0 0x00000000   *................*
0xbffffe5c: 0x00000000 0x00000003 0xbffffeb0 0x00000000   *................*
0xbffffe6c: 0xbffffebb 0x00000000 0xbffffebd 0xbffffec8   *................*
0xbffffe7c: 0xbffffed8 0xbffffee7 0xbffffef1 0xbffffefe   *................*
0xbffffe8c: 0xbfffff30 0xbfffff4a 0xbfffff8d 0xbfffffd4   *0...J...........*
0xbffffe9c: 0x00000000 0x000003e9 0xbfffffe6 0x00000000   *................*
0xbffffeac: 0x00000000 0x6c6c6568 0x2d00326f 0x32007a73   *....hello2.-sz.2*
0xbffffebc: 0x52455400 0x74783d4d 0x006d7265 0x454d4f48   *.TERM=xterm.HOME*
0xbffffecc: 0x6f682f3d 0x722f656d 0x00617961 0x4c454853   *=/home/raya.SHEL*
0xbffffedc: 0x622f3d4c 0x632f6e69 0x55006873 0x3d524553   *L=/bin/csh.USER=*
```

DISPLAY *(cont.)*

DISPLAY *(cont.)*

9. Display pc (fir) register when **threads** is set to “on”:

```
(all:0) > disp -fir

(0:0) ========================================================
 ***** (0:0) thread 0*****
fir      0x0002f140

 ***** (0:0) thread 1*****
fir      0x0001de04

 ***** (0:0) thread 2*****
fir      0x0001a348


(1:0) ========================================================
 ***** (1:0) thread 0*****
fir      0x000108a4

 ***** (1:0) thread 1*****
fir      0x00020dac

 ***** (1:0) thread 2*****
fir      0x00010c5c
```

10. Display local (stack) variable when **threads** is set to “on”:

```
(1:0) > disp i

(1:0) ========================================================
 ***** (1:0) thread 0*****
 ** sigbus_inhrecv.c{}main()#25 i **
  i = 0

 ***** (1:0) thread 1*****
 ** nx_port.c{}n_nx_port_recv() i **  Not found

 ***** (1:0) thread 2*****
 ** sigbus_inhrecv.c{}myhandler()#70 i **
  i = 0
```

See Also

assign, type

EXEC EXEC

Read and execute IPD commands from the specified file.

Syntax

exec [**-echo** | **-step**] *filename*

Arguments

-echo Causes the IPD commands in the specified file to be echoed to the terminal before they are executed. Along with the command, the current prompt is echoed to show the default context. By default, IPD does not echo commands.

-step Causes the IPD command file to be executed line by line. The screen displays each IPD command before executing it (comment lines and blank lines are skipped). Execute the displayed command by pressing `<Enter>`; the next command then appears on the screen. To stop stepping through the command file, use the keyboard interrupt (entering `<Del>` or `<Ctrl-C>`) to terminate the **exec** command.

filename The name of the IPD command file.

Description

The **exec** command specifies an IPD command-file to execute.

When you specify **-echo**, a "++" is prefixed to each command line as it is displayed to show that it is being read from a command file.

You may use the **exec** command inside the command file. Up to eight levels of **exec** nesting are possible. For every nested **exec** level two additional "++" characters are prefixed to command lines that are echoed.

You may insert comments in command files by typing # followed by a space and the comment. All characters, including semicolons, in the line are part of the comment.

IPD executes the commands in a file named *.ipdrc* in your home directory when it starts. The *.ipdrc* file is often used to define configuration information, such as a list of convenient aliases and command line variables. The commands in *.ipdrc* are not echoed.

EXEC *(cont.)*

EXEC *(cont.)*

Examples

1. Execute the command file *picf*, which consists of the following lines:

```
load main -on 0 \; node -on 1..3
context (1..3:0)
break #84
break #90
```

When you execute this file, you see the following display:

```
ipd> exec -echo picf
ipd> ++ load main -on 0 \; node -on 1..3
       *** reading symbol table for main... 100%
       *** loading program...
       *** initializing IPD for parallel application...
       *** reading symbol table for node... 100%
       *** load complete
(0:0)> ++ context (1..3:0)
(1..3:0) > ++ break #84
(1..3:0) > ++ break #90
(1..3:0) >
```

EXIT EXIT

Terminate a debug session and exit IPD.

Syntax

exit

Arguments

None

Description

The **exit** command terminates an IPD session. It is equivalent to the **quit** command. Both commands terminate only those processes that the debugger has loaded.

Examples

1. Exit IPD:

```
(all:0) > exit
  *** IPD exiting...
```

See Also

quit

FLUSH FLUSH

Set performance monitoring instrumentation flush policy.

Syntax

Change performance monitoring event trace buffer **flush** policy:
flush [*context*] [**-stop** | **-wrap** | **-continue**]

List current flush policy:
flush [*context*]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-stop

When the event trace buffer is full, stop performance monitoring, but do not flush the buffer. The application continues to execute with no further data captured. Data for only the first part of the programs execution is captured in the event trace buffer.

-wrap

When the event trace buffer is full, do not flush the buffer but continue to collect performance data by overwriting the oldest event traces (the buffer is used as a circular buffer). Data for only the last part of the programs is captured in the event trace buffer.

-continue

When the event trace buffer is full, flush the buffer and continue to collect performance data. All data is captured in the event trace buffer. Flushing the buffers may adversely affect the performance program.

FLUSH *(cont.)*

FLUSH *(cont.)*

Description

The **flush** command sets the instrumentation flush policy. The flush policy determines how instrumentation data is handled when the internal buffers are filled.

The default flush policy is **-continue**. Refer to the description of the **instrument** command for detailed information about collecting data.

When the **flush** command is used without parameters or only with a context parameter, information on the instrumented flush policy is displayed. An example is as follows:

```
Type       Buffer Size Flush Location  Flush Policy Output File Context
====       =========== ==============  ============ =========== =======
prof       000064       entry of exit   stop         mon.out     (0:0)
paragraph  000064       entry of exit   wrap         pg.trf      (1..3:0)
```

The type of instrumentation is shown at the left, followed by the start and stop locations. Under the "Flush Policy" header is the flush policy: "stop," "wrap" or "continue." At the right is the context of nodes and process types being monitored.

The **flush** command must be performed after the **instrument** command. If the **flush** command is used on a context that is not instrumented, IPD displays just the header.

This command may not be used while examining core files.

Examples

1. Starting from the Unix shell we want to collect paragraph data on the application, *my_app*. The **flush** command is used to change the flush policy to only capture the last part of the *my_app* execution. Here, the event trace buffer is used as a circular buffer:

```
ipd > load my_app
 *** reading symbol table for /home/myacct/my_app... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
(all:0) > instrument paragraph
(all:0) > flush -wrap
(all:0) > run
```

See Also

instrument

FRAME FRAME

Display the call stack traceback(s) of the current or specified context.

Syntax

frame [*context*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

Description

The **frame** command displays a stack traceback, which lists the routines accessed and the files in which those routines are located. If the routine was compiled to produce debug information, line numbers are displayed. If not, memory addresses are displayed.

If IPD encounters a routine without a recognizable function prologue, such as some assembly routines, it assumes a standard stack frame and prints a question mark before the function name to indicate that assumption. This indicates that some functions may be omitted from the stack frame traceback.

Parentheses (( )) following a name indicate a routine. Braces ({ }) indicate a file.

C++ functions inlined by the compiler (via the **-Minline** switch) are not displayed.

The **frame** command is affected by the setting of the **threads** command. If it is set to "on" a stack traceback is displayed for each user thread in each process.

FRAME *(cont.)*

FRAME *(cont.)*

Examples

1. In the following example, execution stopped after the program hung up. The **frame** command traces the stack to provide a history of the routines called, starting from the most recent. In this example, node 3 is found to have a different history than nodes 0, 1, and 2:

```
(all:0) > frame
  ***** (0..2:0) *****
    __flick()    [_flick.s{}0x00023fe8]
    _gdhigh()    [_gdhigh.c{}0x000240f8]
    gdhigh_()    [gdhigh_.c{}0x0001e9dc]
    gauss()    [gauss.f{}#72]
    main()    [pgfmain.c{}0x000001ac]
  ***** (3:0) *****
    __flick()    [_flick.s{}0x00023fe8]
    msgwait_()    [msgwait_.c{}0x0002011c]
    shadow()    [gauss.f{}#209]
    gauss()    [gauss.f{}#58]
    main()    [pgfmain.c{}0x000001ac]
```

2. This example shows how the **frame** command might appear for a C++ application with member functions:

```
(all:0) > frame

     ***** (all:0) *****
       queen::next()    [queen.C{}#67]
       queen::test(int)    [queen.C{}#54]
       queen::first()    [queen.C{}#45]
       queen::first()    [queen.C{}#45]
       queen::first()    [queen.C{}#45]
       queen::first()    [queen.C{}#45]
       main()    [queen.C{}#87]
```

FRAME *(cont.)*

FRAME *(cont.)*

3. In the following example, a breakpoint was set in *main()* at line #21 and a second breakpoint was set in the function *myhandler()* at line #62. The function *myhandler()* is passed as an argument to the *hrecv()* call made in *main()*. If **threads** has been set to *off*, frame produces the following output after running the application to the breakpoints.

```
(0,1:0) > frame
  ***** (0:0) *****
    main()     [hrecvtst.c{}#21]
    _crt0_start()     [crt0.c{}0x000101fc]
  ***** (1:0) *****
    myhandler()     [hrecvtst.c{}#62]
    _nx_port_recv_thread()     [nx_port.c{}0x00017d30]
```

The **process** command shows us that process (1:0) was not running the main user thread when it hit the breakpoint. This is indicated by the ">" in the far left column:

```
(0,1:0) > process
        Context             State      Reason    Location     Procedure
  ===================== ============ ========= ========== ====================
 *(0:0)                 Breakpoint   C Bp 1    Line 21    main()
>*(1:0)                 Breakpoint   C Bp 2    Line 62    myhandler()
```

To see where the main user thread was when the breakpoint was hit, set **threads** to *on* and then review the **frame** command output. In process (0:0), thread 0 is at the breakpoint set in *main()* at line #21. Its other threads are sitting in *_mach_msg_trap()*, which is their normal location when not doing anything. In process (1:0), the main user thread (thread 0) was executing a *printf()* call when the process was stopped by thread 1 when it encountered the breakpoint at line #62 in *myhandler()*. Thread 2 was not doing anything when the process stopped, just as in process (0:0).

```
(0,1:0) > frame
  == (0:0) ==================================================
  ***** (0:0) thread 0 *****
    main()     [hrecvtst.c{}#21]
    _crt0_start()     [crt0.c{}0x000101fc]
  ***** (0:0) thread 1 *****
    _mach_msg_trap()     [unknown1.s{}0x00025a48]
  ? mach_msg()     [mach_msg.c{}0x000257dc]
    mach_msg_receive()     [mach_msg_receive.c{}0x00025950]
    _nx_port_recv_thread()     [nx_port.c{}0x00017a74]
  ***** (0:0) thread 2 *****
    _mach_msg_trap()     [unknown1.s{}0x00025a48]
  ? mach_msg()     [mach_msg.c{}0x000257dc]
    mach_msg_receive()     [mach_msg_receive.c{}0x00025950]
    _nx_port_recv_thread()     [nx_port.c{}0x00017a74]
```

FRAME *(cont.)*

FRAME *(cont.)*

```
 == (1:0) =================================================
  ***** (1:0) thread 0 *****
  ? _write()    [write.s{}0x0002f2f4]
    _xflsbuf()    [flsbuf.c{}0x00029850]
    fwrite()    [fwrite.c{}0x00035534]
    _doprnt()    [doprnt.c{}0x0002f990]
    printf()    [printf.c{}0x0002d8bc]
    main()    [hrecvtst.c{}#29]
    _crt0_start()    [crt0.c{}0x000101fc]
  ***** (1:0) thread 1 *****
    myhandler()    [hrecvtst.c{}#62]
    _nx_port_recv_thread()    [nx_port.c{}0x00017d30]
  ***** (1:0) thread 2 *****
    _mach_msg_trap()    [unknown1.s{}0x00025a48]
  ? mach_msg()    [mach_msg.c{}0x000257dc]
    mach_msg_receive()    [mach_msg_receive.c{}0x00025950]
    _nx_port_recv_thread()    [nx_port.c{}0x00017a74]
(0,1:0) >
```

See Also

disassemble, list

HELP HELP

Display IPD commands and syntax.

Syntax

List all commands:
{ **help** | **?** }

Obtain syntax help:
{ **help** | **?** } *command*

Arguments

command	The *command* argument is any IPD command. The command line syntax will be displayed for the command, with a brief description.

Description

The **help** command displays a summary list of IPD commands, the shortest abbreviation for each command, and a brief description.

A sub-set is displayed if core files are loaded. Only commands that are usable on core files are shown.

HELP *(cont.)*

HELP *(cont.)*

Examples

1. Display the help for the **break** command. (Entering **help break** produces the same result.)

```
(all:0) > ?break
 Display Breakpoint information:
     break [context] [-full]

 Set Breakpoint at procedure:
     break [context] [file{}][class::[class]...]procedure([param-types])
           [-after count]

 Set Breakpoint at source line number:
     break [context] #line [-after count]
     break [context] file{}#line [-after count]
     break [context] [file{}][class::[class]...]procedure([param-types])
           #line [-after count]

 Set Code Breakpoint at instruction address:
     break [context] address [-after count]

The break command allows you to set code breakpoints or display current
breakpoints. The count value specifies the number of times the breakpoint is
encountered before the break occurs (default is 1). Breakpoints are defined in
the current context (either default or specified).

This command is not allowed when examining core files.
```

HELP *(cont.)*

HELP *(cont.)*

2. Display the IPD command summary list. Entering ? produces the same result.

```
(all:0) > help
Commands are grouped functionally.  The information provided for each command
is: command name, shortest acceptable abbreviation, and brief description.

Enter 'help <command>' to get detailed information for each command.

Program load and termination:
  load          loa     Load programs
  coreload      cor     Load core files for examination
  kill          k       Terminate processes

Program execution control and state:
  continue      conti   Continue stopped processes
  run           ru      Start process execution from the beginning
  rerun         rer     Same as run, but do not reuse previous argument list
  process       p       Display the current state of processes
  frame         fr      Display stack traceback from current execution point
  stop          sto     Halt process execution
  wait          wai     Wait for processes to stop
  break         b       Set or display breakpoints
  trace         tr      Set or display tracepoints
  watch         wat     Set or display watchpoints
  remove        rem     Remove break/trace/watchpoints
  step          ste     Execute the next source statement or instruction
  signal        si      Modify or display IPD signal reporting

Program performance analysis data collection:
  instrument    i       Instrument program for collecting performance data
  flush         fl      Set flush policy for event trace buffers

Program data display and modification:
  assign        as      Assign a new value to a variable or address
  display       disp    Display the value of an expression, address, or register
  type          ty      Display the type of an expression

Program message queue display:
  msgqueue      ms      Display the queue of messages sent but not received
  recvqueue     rec     Display the queue of receives posted but not satisfied

Program code display:
  list          li      List source code
  disassemble   disa    Display an assembly listing of program code
  source        so      Set or display source directory search paths
```

HELP *(cont.)*

HELP *(cont.)*

```
IPD control and information:
  context      conte   Change default list of processes to apply commands to
  commshow     com     Display communicator handles assigned by debugger
  exec         exe     Read debugger commands from a file
  exit         exi     Exit IPD - same as quit
  help or ?    h       Display command summary or details
  log          log     Record the debug session in a file
  more         mo      Turn terminal scrolling on or off
  msgstyle     msgs    Set or display context format
  alias        al      Set or display command aliases
  unalias      una     Delete aliases
  set          se      Set or display debug variables
  unset        uns     Delete debug variables
  status       sta     Display version number and control values
  system or !  sy      Execute a UNIX shell command
  threads      th      Set or display threads mode
  quit         q       Exit IPD - same as exit
```

INSTRUMENT INSTRUMENT

Add, remove, or display program instrumentation for performance monitoring.

Syntax

Instrument program:
instrument [*context*] [[**-on**] *perf_name* [*start_location* [,*stop_location* [,*write_location*]]] [**-bufsize** *value*][[**-force**] *path_name*]]

Immediately write performance data and terminate monitoring:
instrument [*context*] **-write**

Remove performance monitoring instrumentation:
instrument [*context*] **-off** [**-nowrite** | **-write**] [*perf_name*]

List performance monitoring instrumentation information:
instrument [*context*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-on Turn profile instrumentation on. This is the default action when the *perf_name* switch is given without **-on** or **-off**.

-off Stop collecting data and remove all profile instrumentation.

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

-write

When used without the **-off** switch, the **-write** switch manually simulates a *write_location*. Immediately write all performance data to the *path_name* directory or file and terminate performance monitoring. The code is still instrumented but no further performance data will be collected. If the program is rerun then new performance data will be collected. The **-write** switch is used to obtain performance data when the application never executes the *write_location* (such as a dead-lock, an infinite loop or program fault).

When used with an **-off** switch the **-write** switch causes IPD to write all performance data before removing instrumentation.

-nowrite

When using the **-off** switch to turn off performance monitoring, the **-nowrite** switch specifies that no performance data should be written.

perf_name

perf_name is the name of the performance utility to be used to analyze the resulting performance data. The three performance utilities are *prof*, *gprof* and *paragraph*, corresponding to the switches **-prof**, **-gprof** and **-paragraph**.

start_location

The *start_location* is the point in the code at which profiling begins. This can be an entry or exit point to a procedure, a line number, or an address. The syntax for the *start_location* specification is one of the following:

[**-entry**|**-exit**] [*file*{}]*procedure*() ([*type* [,*type*]...])

[**-entry**|**-exit**] [*file{}*][*procedure*()]#*line*

[**-entry**|**-exit**] *address*

stop_location

The location at which performance data collection ends. The *stop_location* can be an entry or exit point of a procedure, a line number, or an address. The syntax for the *stop_location* specification is one of the following:

[**-entry**|**-exit**] [*file*{}]*procedure*() ([*type* [,*type*]...])

[**-entry**|**-exit**] [*file{}*][*procedure*()]#*line*

[**-entry**|**-exit**] *address*

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

write_location The location at which all performance data is written and performance monitoring is terminated. The *write_location* can be an entry or exit point of a procedure, a line number, or an address. The syntax for the *write_location* specification is one of the following:

[**-entry**|**-exit**] [*file*{}]*procedure*() ([*type* [,*type*]...])

[**-entry**|**-exit**] [*file{}*][*procedure*()]#*line*

[**-entry**|**-exit**] *address*

Syntax elements for *start_location*, *stop_location*, and *write_location* are defined as follows:

file The name of the source module in which the procedure or line resides. To refer to a line in a file other than the file containing the location of the current execution point, prefix the line number with *file*. When you refer to a procedure, you may omit the *file* name unless there are duplicate procedure names that require qualification.

procedure The name of the procedure at which you want to set the start, stop, or write location, or the precedure in which the line you are specifying resides. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class:: is the name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

INSTRUMENT *(cont.)* INSTRUMENT *(cont.)*

type is any legal C++ type specification, such as *int*, *float* *, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

#line The source line number at which you want to set the start, stop, or write location. The line number must be preceded with a pound sign (#). The statement must be executable. For example, you cannot set a start, stop, or write location on a Fortran **FORMAT** statement, a comment, or an empty line.

address The address at which you want to set a start, stop, or write location. The address must be an instruction address.

-entry Place a start, stop, or write location at the entry of the procedure specified by *procedure*(). This is the default action when only a *procedure*() name is given.

-exit Place a start, stop, or write location at the exit of the procedure specified by *procedure*().

-bufsize *value* The **bufsize** switch specifies the size of the performance monitoring trace buffer in K bytes. Valid numbers are 1 to the maximum amount of memory supported by the system. The default buffer size is 64, or 64K bytes. For **prof** and **gprof** instrumentation, the **bufsize** parameter is ignored.

-force The **-force** switch forces the deletion of the file or directory specified by the *path_name* parameter.

NOTE

You will not be queried if the file or directory exists if **-force** is used.

For added safety, the **-force** switch cannot be abbreviated.

INSTRUMENT *(cont.)* INSTRUMENT *(cont.)*

path_name

path_name can either be a single event trace file for all nodes and processes or *path_name* can be a directory name where a data file exists for each process in the application. If two instrument commands are used to instrument two different contexts, the p*ath_name* can be the same if the *perf_name* switch is the same. However, if the *perf_name* switch is different then an error message is displayed.

For **paragraph** instrumentation, *path_name* is the name of the file that will contain event trace information for all nodes and processes sorted in time order. The default **paragraph** *path_name* is *./pg.trf.* If the *path_name* parameter specifies a directory then the performance data file is *path_name*. If *path_name* already exists and the **-force** switch has not been used, you are queried before the file is overwritten. This query only occurs on the first instrument **-paragraph** command since multiple paragraph instrument commands are supported on different contexts.

For **prof** and **gprof** instrumentation, the *path_name* specifies a directory where the performance data files are written. The individual data files for each process are written to a file named *executable_name.pid.node.ptype* where *pid* is the process id, *node* is the node number, *ptype* is the current process type at the time the data is written to the file. Node and *ptype* follow the IPD naming convention. The default **prof** instrumentation *directory_name* is *mon.out,* while the default **gprof** instrumentation *directory_name* is *gmon.out.* If *path_name* exists and the **-force** switch has not been used, the you are queried before its removal. The query is performed on the first set of **prof** or **gprof** instrument commands.

The **prof** and **gprof** directories contain an auxiliary file named **INFO**, which contains information on each of the data files. The **INFO** file has the following format:

Controlling Process: *executable_name pid_value*

pid	node	ptype	Executable
xxxxxxxxxx	*xxxx*	*xxxx*	*full_path*
....			
....			

The first line has three fields, the title, the name of the controlling process's executable, and its process id. The second line contains column titles for the lines to follow. Each of the rest of the lines, (line 3 through the last line in the file) contain a 10-character process id in the first field, a four-character node number in the second field, a four-character process type number in the third field and the full path name of the executable file for the process in the last field.

INSTRUMENT *(cont.)* INSTRUMENT *(cont.)*

Description

The **instrument** command starts and stops performance data collection for tools such as **prof, gprof,** and **paragraph.**

The instrumentation of the code occurs when the instrument command is given. The actual collection of the data occurs when the program is executed. Data collection starts at the *start_location* and ends at the *stop_location.* The start and stop locations can be placed inside of a loop to monitor the program only when it is executing within the loop. At the *write_location* all performance data is written and performance monitoring terminated, however the instrumented code still exists in the application. This allows you to rerun the program to obtain additional performance data. Use the instrument **-off** switch to remove the performance monitoring instrumentation.

After the data has been written, a completion message is displayed. If no data was collected, this message will be followed by a message informing you that no data was written. The default *write_location* is the *stop_location.* If only the *start_location* is specified, performance monitoring starts at that location and continues until the end of the program. If neither start nor stop are specified, performance monitoring begins at the current execution point and continues until the end of the program. If the *write_location* is encountered before the *start_location* no performance data is generated. If the program does not execute the *write_location* for any reason (such as a dead-lock, infinite loop or program fault) use the **-write** switch to obtain whatever performance data was collected.

The instrument command re-applies the instrumentation when an application is rerun.

All nodes and processes executing this command must be running the same load module. If not, the command returns an error. Multiple **instrument** commands are used to instrument different load modules where differing start, stop or write locations are desired. You may not specify **prof, gprof** and **paragraph** instrumentation on the same context. However, you may instrument one context with **prof**, a second context with **gprof** and a third context with **paragraph.**

Use the **flush** command to modify the performance monitoring trace buffer flush policies when **paragraph** performance monitoring is specified.

To analyze the data generated by the **prof** instrumentation use the **prof** utility. By default, the **prof** utility uses the data in the **INFO** file of the *mon.out* directory to choose the lowest (node:ptype) pair data file for the specified load file. To view **prof** output on other (node:ptype) pairs, specify the *executable_name.pid.node.ptype* data file via the **prof** utility **-m** switch.

To analyze the data generated by the **gprof** instrumentation use the **gprof** utility. By default, the **gprof** utility uses the data in the **INFO** file of *gmon.out* directory to choose the lowest (node:ptype) pair data file for the specified load file. To view **gprof** output on other (node:ptype) pairs, specify the *executable_name.pid.node.ptype* data file on the **gprof** utility command line.

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

Use the **paragraph** utility to analyze the data generated by the **paragraph** instrumentation. By default, the **paragraph** utility uses the default *pg.trf* file. If you have used a different file name when instrumenting **paragraph**, use the **paragraph -f** switch or the **paragraph** file menu to specify your data file.

The application must be compiled with the **-Mperfmon** switch (the default) in order for the performance library to be linked into the application. If the application is compiled with the **-Mnoperfmon** switch, the **instrument** command will not be able to instrument the application.

When the **instrument** command is started without parameters, or only with a context parameter, information on what has been instrumented is displayed:

```
Type        Start Location   Stop Location    Status     Context
===         ==========       ==========       ====       ======
prof        test{}#23        entry of exit()  applied    (0:0)
paragraph   main{}52         entry of exit()  applied    (1..3:0)
```

The type of instrumentation is shown at the left followed by the start and stop locations. Under the Status header is the status of the instrumentation: "applied", "monitoring" or "done." The "applied" status indicates that the instrumentation is applied but the application is not running. The "monitoring" status indicates that the application is running and performance monitoring is occurring. Finally at the right is the context of nodes and process types being monitored.

This command may not be used while examining core files.

Examples

1. Starting from the Unix shell, **prof** performance data is gathered on an application, *my_app*, for its entire run:

```
ipd
ipd > load my_app
 *** reading symbol table for /home/myacct/my_app... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
(all:0) > instrument -prof
(all:0) > run
```

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

2. After starting IPD and loading the program, **gprof** data starts collecting at the next call of the function **my_func()**. The output directory is specified to be *gprof_data* so that current data in the *gmon.out* directory is not overwitten:

```
(all:0) > instrument -gprof my_func() gprof_data
(all:0) > conti
```

3. For the program shown below a paragraph event trace for the do loop is generated. After starting IPD and loading the program, the data collection is started with the instrument command shown. This collects data on the program while it is in the loop and writes the data to the default paragraph *pg.trf* file. Note that the stop location is placed outside the do loop:

```
005          program main
006
007          call init
008          do 10 i=1,n
009            ...
010  10    continue
011            ...
012            end
```

```
(all:0) > instrument -paragraph #8,#11
(all:0) > conti
```

4. For the program shown below **gprof** data is generated only when the program is executing in the inner do loop of the subroutine calculate:

```
005          subroutine calculate
006
007          do 5 j=1,5
008              do 10 i=1,20
009                ...
010  10          continue
011  5       continue
012            ...
013            end
```

```
(all:0) > instrument -gprof #8,#10,#13
(all:0) > conti
```

Here, the start and stop locations are specified within the do loop but the write location is specified outside of the outer loop. By specifying the write location outside the loop, only when the program is in the inner loop of the subroutine calculate is **gprof** performance data collected.

INSTRUMENT *(cont.)*

INSTRUMENT *(cont.)*

5. There are two different load modules: my_host on node 0 and my_nodes on nodes 1 through 3. This command line gathers **gprof** data on my_host and **paragraph** data on my_nodes:

```
ipd > load my_host -on 0 \; my_nodes -on 1,2,3
 *** reading symbol table for /home/myacct/my_host... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
 *** reading symbol table for /home/myacct/my_nodes... 100%
 *** load complete
(all:0) > instrument (0:0) -gprof
(all:0) > instrument (1..3:0) -paragraph
(all:0) > run
```

See Also

flush

KILL KILL

Terminate and remove processes in the current or specified context.

Syntax

kill [*context*] **[-force] [-fault | -nonfault | -notfirst]**

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-force Kill process(es) without asking for verification.

-fault The **-fault** switch removes faulting processes from core analysis. This switch is for use only during core-file analysis.

-nonfault The **-nonfault** switch removes non-faulting processes from core-file analysis. This switch is only for use during core-file analysis.

-notfirst The **-notfirst** switch removes all processes from core-file analysis except for the one that faulted first, as determined by the internal time stamp. This switch is for use during core-file analysis only.

Description

The **kill** command terminates and removes processes. Because the **kill** command is potentially destructive, it asks if you are sure you want to kill the processes. You must enter a *y* to confirm that you want to kill the process. Any other character(s) are interpreted as a "no". Use the **-force** switch to force the kill without a confirmation.

KILL *(cont.)*

KILL *(cont.)*

The **-force** switch is not necessary when executing a command file. IPD automatically suppresses the confirmation message when reading commands from a file.

A **kill** command terminates a process and destroys all information IPD has about the process, including breakpoints, variable types, etc. When all processes in the default context have been killed, the prompt reverts to "ipd >" to indicate the lack of a current context.

If a core file is being analyzed, the **kill** command removes processess' core files from analysis. A *context* argument may specify processes, or the **-fault**, **-nonfault**, or **-notfirst** switches may be used to remove groups of core files. If all processes' core files are removed (the context is emptied) the core-file-analysis mode is ended.

Examples

1. Kill process 0 on node 0 when the current context is *(1..3:0)*:

```
(1..3:0) > kill (0:0)
     ***This command will delete all processes in (0:0).
     Are you sure you want to do this(y/n)? y
(1..3:0) >
```

2. Kill all processes in the current context without a question.

```
(all:0) > kill -f
ipd >
```

KILL *(cont.)*

KILL *(cont.)*

3. All processes of an application that faulted were loaded initially for core-file analysis. Now, only the faulting processes are of interest, so the rest are removed from analysis, leaving nodes 0..2 in the context. Then, to exit the core-file-analysis mode completely, the remaining processes are killed:

```
ipd > coreload -all
 *** reading symbol table for /home/john/chess/cgame.nx... 100%
 *** scanning core files...
 *** coreload complete
 Context  State    Reason    Location     Procedure
 =======  =====    ======    ========     =========
*(0..2:0) Signaled SIGBUS    Line 113     nextmove()
*(3:0)    Signaled SIGKILL   Line 345     forward()
*(host)   Signaled SIGKILL   0x00010404   nx_wait()
(all:0) >
.
.
.
(all:0) > kill -nonfault
(0..2:0) > process
 Context  State    Reason    Location     Procedure
 =======  =====    ======    ========     =========
 (0..2:0) Signaled SIGBUS    Line 113     nextmove()
(0..2:0) >
.
.
.
(0..2:0) > kill
ipd >
```

See Also

stop

LIST LIST

Display source code lines.

Syntax

List source code from current execution point:
list [*context*] [,*count*]

List source code starting from a procedure:
list [*context*] [*file*{}]*procedure*() [,*count*]

List source code starting from a source line number:
list [*context*] [*file*{}] [*procedure*()] *#line* [**:** *#line* | ,*count*]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

count

The *count* argument is an integer that indicates the number of lines of source code to list. If the *count* argument is positive, listing starts at the specified location and continues for the specified number of instructions. If negative, listing begins at *count*-1 instructions preceding the specified location and ends at that location.

If you do not specify a *count* argument, IPD uses the last *count* argument supplied to a **list** command in the current session, except when listing an entire procedure. The default value of the *count* argument is 50 lines. One way to use the *count* argument is to specify a large value and then use the IPD **more** utility (see the **more** command) to browse through the instructions.

LIST *(cont.)* LIST *(cont.)*

file

The *file* argument is the name of the source file in which the procedure or line resides. To refer to a file other than the location of the current execution point, you must prefix the line number with *file*. When you refer to a procedure, you may omit the file name unless there are duplicate procedure names, because IPD can find the source file from the symbol table information. The *file* argument must be followed with a pair of braces ({}).

procedure

The optional *procedure* argument is the name of the procedure at which you want to start listing or the procedure in which the line number you are specifying resides. You need to specify the procedure when the execution point is not in the same procedure as the variable. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class::

The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

type

Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

line

The *line* argument specifies a source line number at which to start listing. The line number must be prefixed by a number sign (#) and may be any source line in a source file. You may specify a range of lines by specifying a begininng and ending line number (you must specify the range in ascending order), or or by specifying a line number and a line count.

LIST *(cont.)* LIST *(cont.)*

Description

The **list** command displays source code lines. To list from the current execution point, all processes in the context must be stopped. This is because the current execution point can be defined only when all processes in the context are stopped. If you specify a line number or procedure as the starting point, the state of the processes does not matter.

IPD finds the source files by searching the source directory search path defined by the **source** command. You may not specify a path as part of the *file*{} qualifier. Refer to the **source** command for more information.

If you enter the command without specifying a starting point (using the current execution point), the **list** command lists the source lines at the current execution point. If the default or specified context has processes stopped at different locations, multiple listings are displayed, one for each process with a unique execution point.

If you specify a *file*{}, it must have been used in compiling of a loaded module. Source files unrelated to any loaded module cannot be listed with the **list** command. Use the **system** command to access operating system commands such as **cat** or an editor to look at other files. Specifying a source file that has the same name as a file used in compiling a program under debug, but is not the actual file used does not generate an error or warning, but may provide faulty information. There is no way for the debugger to detect this circumstance.

If you specify a *procedure()* argument without *count* or *#line* arguments, then the entire procedure is listed, regardless of the last value of *count* specified.

Before each listing, the **list** command displays a line showing the current context and the name of the source file that is being listed. If the source lines being listed are from a file that does not contain a current execution point, the context information is omitted, and only the file name is displayed prior to the listing.

Line numbers that are valid for setting breakpoints, tracepoints, and watchpoints are followed by the ">" symbol. All other line numbers are followed by the ":" symbol. A valid line number is one that is represented in the line table created when an application is compiled with **-g**. However, the number of lines represented in a line table is reduced as the optimization level (specified using the **-O** compiler switch) increases.

LIST *(cont.)* LIST *(cont.)*

Examples

1. Assume that the current context is (1:0). Issue the **list** command after the main program encounters a code breakpoint to display each source line you are stepping through:

```
(1:0) > run ; wait
 Context      State        Reason    Location         Procedure
 =======      =====        ======    ========         =========
*(1:0)        Breakpoint   C Bp 1    Line 180         shadow()

(1:0) > step ; list,1
 Context      State        Reason    Location         Procedure
 =======      =====        ======    ========         =========
*(1:0)        Stepped                Line 180         shadow()
  ***** (1:0) *****
  File: ./gauss.f
180> if(iam.eq.0) then

(1:0) > step ; list
 Context      State        Reason    Location         Procedure
 =======      =====        ======    ========         =========
*(1:0)          Stepped              Line 194         shadow()
  ***** (1:0) *****
  File: ./gauss.f
194> leftid = irecv(type, a(1,1), length)
```

LIST *(cont.)*

LIST *(cont.)*

2. List 17 source lines starting at line number 180 (entering ***list #180,17*** would produce the same result):

```
(1:0) > list #180:#196
180>    if(iam.eq.0) then
181:c
182:c   If I am the leftmost node of the array (node 0) then only exchange
183:c   with the right (to the left is a boundary of the array)
184:c
185>         rightid = irecv(type, a(1,range+2), length)
186>         call csend(type, a(1,range+1), length, rightnode,0)
187>         call msgwait(rightid)
188:
189:      else if (iam .eq. nbrnodes) then
190:c
191:c   If I am the rightmost node of the array (highest numbered node) then
192:c   only exchange with the node to the left.
193:c
194>         leftid = irecv(type, a(1,1), length)
195>         call csend(type, a(1,2), length, leftnode,0)
196>         call msgwait(leftid)
(1:0) >
```

See Also

source, disassemble

LOAD LOAD

Load an application under debugger control.

Syntax

load *filename* [<*infile*] [> *outfile*] [*program_args*]

Arguments

filename The *filename* argument is the program that you want to load. Specify the path name if the file is not in the current directory.

infile The *infile* argument is the program's input file. All of the program's standard input (stdin) is read from *infile*. The *infile* is not read until a **wait** command is issued.

outfile The *outfile* argument is the program's output file. All of the program's output will be redirected to the *outfile*.

program_args These are arguments that are passed to the program. Anything following the file-redirection arguments, up to the end of the line, a non-escaped pound sign, or a non-escaped semicolon, is used as a program argument.

If the program is compiled with the **-nx** option, arguments should include any operating system command line arguments necessary for loading the application (such as **-pn** *partition*, **-sz** *num_nodes*, **-pt** *process_type*, **-nd** *node_list*, and so on). For a complete description of these arguments, see the *Paragon™System User's Guide*.

Description

The **load** command loads an application under the debugger's control and sets the default debug context. The *program_args* arguments may include those special switches recognized by the **-nx** runtime start-up routine, such as **-sz**, **-pn**, and so on.

To include the special characters ";", "#", "$", or "\" as arguments, they must be escaped (preceded) with a backslash character ("\").

The **run** and **rerun** commands may be also used to specify command line arguments or to redirect standard input. Those commands cause the application to be reloaded.

LOAD *(cont.)*

LOAD *(cont.)*

The **load** command may be used to load different programs on different nodes. To do so, compile the application with the **-nx** switch and specify the additional programs in the argument list as described in the *Paragon™System User's Guide.*

The load command sets the default context. For parallel applications compiled with **-nx**, the default context is automatically set to include all compute processes that have the same ptype as the first program specified on the command line. For all other applications, the **load** command is set to the default context (host).

Programs that call **nx_nfork()** or **nx_loadve()** directly may cause other processes to be loaded when they are executed. When the new processes are created, the IPD program prints an information message. At that point, the new processes are available for debugging by changing context.

Examples

1. Load the file *gauss* (compiled with the **-nx** option) on all nodes in the partition named *eldr*; set the process type to 99:

```
ipd > load gauss -pn eldr -pt 99
 *** reading symbol table for /home/myacct/gauss... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
(all:99) >
```

2. Load the file *gauss* on 3 nodes in the default partition; set the process type to 99; redirect input to come from the file *gauss.dat* and pass the program the additional argument "100":

```
ipd > load gauss < gauss.dat -sz 3 -pt 99 100
 *** reading symbol table for /home/myacct/gauss... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** load complete
(all:99) >
```

LOAD *(cont.)*

LOAD *(cont.)*

3. Load the file *gauss1* on node 0 in the default partition and set the process type to 1; load the file *gauss2* on nodes 1..3 in the default partition and set the process type to 2:

```
ipd > load gauss1 -on 0 -pt 1 \; gauss2 -on 1..3 -pt 2
 *** reading symbol table for /home/myacct/gauss1... 100%
 *** loading program...
 *** initializing IPD for parallel application...
 *** reading symbol table for /home/myacct/gauss2... 100%
 *** load complete
(0:1) >
```

4. Load the file *sample* (compiled without the **-nx** option).

```
ipd > load sample
 *** reading symbol table for /home/myacct/sample... 100%
 *** loading program...
 *** load complete
(host) >
```

See Also

coreload, status

LOG LOG

Turn debug session logging on or off, or display the name of the current log file.

Syntax

log [[**-on**] *filename* | **-off**]

Arguments

[**-on**] *filename* Specifies the name of the file to contain the debug log. The *filename* argument may be a complete or relative pathname. The **-on** is optional if you specify a file name.

-off Turns off logging to the current log file.

Description

The **log** command with no arguments displays the name of the current log file. The arguments allow you to specify a log file name and turn on logging, or to turn it off. Only one log file may be active at a time. If IPD is currently using one log file and you use the **log** command to specify another log file, the current log file is closed and the new log file is opened.

If you specify a log file that already exists the file will be overwritten with new log information.

Examples

1. Turn on logging to file *gauss.log*:

```
(all:0) > log gauss.log
```

2. Display the name of the current log file:

```
(0:0) > log
        Log file: gauss.log
```

See Also

status

MORE MORE

Control scrolling of IPD information on the display.

Syntax

more [**-on** | **-off**]

Arguments

-on Turns on the **more** function to control scrolling of the display. Whenever output from a command would scroll off the screen, the display is halted. A **more** prompt is shown below the last displayed line, at the bottom of the screen, and the IPD program waits for input (pressing any key on the keyboard) before continuing.

-off Turns off the **more** function for terminal output. Allows output to scroll freely, even when it is greater than one screen in length.

Description

The **more** command allows you to control information scrolling on the display returned by IPD commands. The default **more** state depends upon IPD's standard input and standard output. If the standard input and standard output are a terminal, then the default is "**more -on**". However, if IPD's standard input or standard output is a file then the default is "**more -off**".

To determine the current IPD **more** state, use the **more** command without arguments.

Examples

1. Turn on IPD's **more** function:

```
(all:0) > more -on
```

2. Display the current **more** state:

```
(all:0) > more
 More: on
```

See Also

status

MSGQUEUE MSGQUEUE

Display messages sent but not yet received.

Syntax

msgqueue [*context*] [*type*] [**-all**]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

type

The user-defined message type as specified in the message-passing call. Only messages of the type specified by the *type* argument will be displayed; otherwise, all types in the context are listed.

-all

Include both NX and MPI style messages in the display.

Description

The **msgqueue** command displays the messages that have been sent and have arrived on the node(s) in the current context but have not yet been received by a process on the node. If you do not specify a type, all message types are included, including those sent by library calls. Use the **recvqueue** command to display the processes that have posted receives that have not been satisfied.

MSGQUEUE *(cont.)* MSGQUEUE *(cont.)*

The **-all** switch is useful for showing MPI messages for all communicators; without it, only MPI messages for the current context's communicator are displayed. The **-all** switch is also useful if an application uses both NX and MPI message passing, for example, if an MPI application is linked with a library using NX message passing. Once an application has executed an *MPI_Init*() call, only MPI-style messages are displayed by default, since in most cases the message passing occurring within the library is not a concern. Since seeing these messages may be useful in understanding a problem, the **-all** switch causes NX messages to be displayed along with the MPI messages.

If the **msgstyle** command is set to NX, all messages (messages sent using MPI calls as well as those sent using NX calls) will be displayed as NX-style messages.

It is possible for a message to be held up on the sending node if its receive has not been posted and there is insufficient memory on the receiving node to allow it to store even a fragment of the message. Such a message will not appear on the **msgqueue** list.

It is also possible for a message for which a receive has been posted to still appear on the message list if the user's receive buffer was paged out when the process was stopped and the queue was requested. This is a temporary state such that the receive has been posted, but before the page containing the receive buffer could be swapped in and the receive completed the user stopped the process. This situation can be detected by displaying the **recvqueue**. If it shows that a receive has been posted for a message that is still on the **msgqueue** list, then the message is in the process of being received.

If a global message is sent (that is, sent with **-1** as the node number), the transfer is optimimized by passing the messages through a tree structure. This message will not be visible on all of the node's **msgqueue** lists as the messages are sent to one level of the tree and then passed to the next.

If a synchronous MPI send (*MPI_Ssend()* or *MPI_Issend()*) is used and the corresponding receive has not been posted, the **msgqueue** list may include a 12-byte intermediate request-to-send message that is sent to the receiver as part of the underlying communication protocol. Typically, message types of 1,000,000,000 or greater are system messages. Refer to the *Paragon(TM) System C Calls Reference Manual*, Appendix A, for more information.

Use the **recvqueue** command to display posted receives that have not been satisfied.

In an MPI application, if the communicator used to send an unreceived message has been freed the communicator name given in the display will be COMMUNKNOWN and the ranks listed will be relative to COMMWORLD.

MSGQUEUE *(cont.)*

MSGQUEUE *(cont.)*

Examples

1. Display all messages sent to process type 0 that have not been received:

```
(all:0) > msgq
 *** Unreceived messages in (all:0)
                                                       Msg Length
 Source                 Destination            Msg Type    (in bytes)
 =================      =================      ==========  ==========
 (0:0)                  (2:0)                           2        7912
 (2:0)                  (3:0)                           1        6048
 (1:0)                  (3:0)                           2        7912
```

2. In an MPI application, display any messages sent within the COMMWORLD communicator that have not been received:

```
(COMMWORLD:all) > msgq
*** Unreceived messages in (COMMWORLD:all)
                                                       Msg Length
  Source                Destination            Msg Tag     (in bytes)
 =================      =================      ==========  ==========
 (COMMWORLD:0)            (COMMWORLD:10)          16           100
 (COMMWORLD:0)            (COMMWORLD:23)          16           100
```

3. In an MPI application that is linked with the ProSolver library (which is written using NX message passing), display all messages that have not been received. Messages sent by any communicator are displayed as well:

```
(COMMWORLD:all) > msgq -all
*** Unreceived MPI messages in (COMMWORLD:all)
                                                     Msg Length
  Source               Destination            Msg Tag     (in bytes)
 =================      =================      ==========  ==========
 (COMMWORLD:0)            (COMMWORLD:10)          16           100
 (COMMWORLD:0)            (COMMWORLD:23)          16           100
 (COMM1:1)                (COMM1:0)                8            80

*** Unreceived NX messages in (all:0)
                                                     Msg Length
  Source               Destination            Msg Type    (in bytes)
 =================      =================      ==========  ==========
 (0:0)                   (2:0)                  200014          16
 (2:0)                   (3:0)                  200015         140
```

MSGQUEUE *(cont.)* MSGQUEUE *(cont.)*

See Also

recvqueue

MSGSTYLE MSGSTYLE

Set or display how process identifiers within contexts are displayed and interpreted.

Syntax

msgstyle [**-nx** | **-mpi**]

Arguments

-nx Use the NX naming convention for process identification (node and ptype).

-mpi Use the MPI naming convention for process identification (communicator and rank).

Description

The **msgstyle** command allows you to specify whether contexts should be displayed and interpreted as node/ptype pairs (NX) or communicator/rank pairs (MPI). When an application is first loaded, NX is assumed. After the execution of *MPI_Init()*, MPI is assumed. This command allows you to force the display of processes using the node/ptype notation while debugging an MPI application.

This command is most useful when debugging an application that mixes NX and MPI styles of message passing. This situation may occur for an MPI application that calls a library that uses NX message passing.

Specifying **-mpi** prior to executing a call to *MPI_Init()* or after a call to *MPI_Finalize()* is ignored.

Using **msgstyle** without an argument displays the current message-display mode, either NX or MPI.

Examples

1. Tell IPD to display contexts using the node/ptype format:

```
(COMMWORLD:all) > msgstyle -nx
(all:0) >
```

See Also

context

PROCESS PROCESS

Display information about user processes controlled by IPD.

Syntax

process [*context*] [**-change**] [**-loadfile**] [**-full**]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-change

The **-change** switch displays only those processes that have changed state since the last process display.

-loadfile

The **-loadfile** switch displays the load module name instead of the source file name. By default, the current source file and procedure are displayed for stopped processes, and the load module name is displayed for running processes. The **-loadfile** switch must be used with the **-full** switch.

-full

The **-full** switch displays the process information in a long or "full" format with more room for file, class, and procedure names.

PROCESS *(cont.)*

PROCESS *(cont.)*

Description

The **process** command provides information about the processes running under IPD. The following is an example of the **process** display:

```
   Context     State        Reason       Location    Procedure
  =========    ======       ======       ========    =========
  (1,2:0)      Breakpoint   C Bp 1       line #53    scan()
* (3..5:0)     Executing
  (11:0)       Breakpoint   C Bp 2       0x000456    test()
```

If an asterisk (*) appears in the first column of the process display, then the processes on that line of the display have changed state since the last process display. The column headings denote the following:

Context — The nodes and process types in context format (see the **context** command for more information). If the "Context" field overflows, the **process** command splits the information into multiple lines.

State — The current state of the processes. A process can be in one of eleven states: "Initial," "Executing," "Breakpoint," "Watchpoint," "Src Stepping," "InstStepping," "Stepped," "Signaled," "Interrupted," "Exited," or "Exiting. "

Reason — For processes in the "Breakpoint," "Watchpoint," "Signaled," or "Exited" states, the next column under the heading "Reason" gives further information on the state of the process. For processes at a breakpoint or watchpoint, the "Reason" column shows the breakpoint/watchpoint type and number (see the **break** and **watch** commands for more information). For terminated processes, this column describes why the process has exited.

Location — Shows the location of the process for all states except "Executing," "SrcStepping," and "InstStepping."

Procedure — Shows the name of the procedure for all states except "Executing," "SrcStepping," and "InstStepping."

Use the **-loadfile** and **-full** switches together to specify that the load module should be displayed instead of the file name and procedure.

PROCESS *(cont.)*

PROCESS *(cont.)*

The **process** command is affected by the use of the **threads** command. When it is set to "off" the display shows information for a single thread in each process as illustrated above. Normally, the thread shown is the main user thread. However, if a thread other than the main user thread is halted for any reason (such as a fault or breakpoint), then the state information for the thread that caused the process to be halted is displayed instead. If this is the case, a ">" is placed in the first column, as in the following display:

```
   Context      State        Reason    Location       Procedure
  =========     ======       ======    ========       =========
>*(0:0)         Signaled     SIGSEGV   0x0001024      hrecv_hndlr()
  (1,2:0)       Breakpoint   C Bp 1    line 102       scan()
```

If **threads** is set to "on", process information is displayed for each user thread in each process. A column labeled "Thrd" is added that displays an identifier for each thread. The thread that caused the process to halt shows the State and Reason information indicating why the process stopped. All other threads will have a Stopped state and an empty Reason field.

```
Context Thrd  State        Reason    Location        Procedure
======= ===== ======       ======    ========        =========
  (0:0)  0   Stopped                 line 84         Main()
         1   Stopped                 0x00017d2c      _nx_port_recv_thr
         2   Breakpoint   C Bp 1     line 70         myhandler()
  (1:0)  0   Breakpoint   C Bp 2     line 10         sub1()
         1   Stopped                 0x00017d2c      _nx_port_recv_thr
         2   Stopped                 0x00017d2c      _nx_port_recv_thr
```

The **wait** and **step** commands perform an implicit **process** command upon returning control to the user.

A process in the "Exited' state no longer exists under debug control.

Examples

1. Display process information. Node 0 is stopped at a breakpoint and the others are in a just-loaded state.

```
(all:all) > process

   Context      State          Reason        Location    Procedure
  =========     ======         ======        ========    =========
  (0:0)         Breakpoint     C Bp 1        line #86    Main()
  (1..3:0)      Initial                      line #84    Main()
```

PROCESS *(cont.)* PROCESS *(cont.)*

2. Continue the execution of the *node* program, hitting another breakpoint. The **wait** command performs an implicit **process** command to display the process information. Notice that the *node* program has executed and is now stopped at a breakpoint. The leading asterisk (*) indicates that the state has changed since the last time **process** was used:

```
(all:all) > context (1..3:0)
(1..3:0) > continue
(1..3:0) > wait
   Context    State          Reason       Location   Procedure
  =========   ======         ======       ========   =========
* (1..3:0)    Breakpoint     C Bp 3       line 93    Main()
```

3. Redisplay the process information in the “full” format:

```
(1..3:0) > process -full
                                     Location
                                     Procedure
   Context    State         Reason   Src/Obj Name
  =========  ======         ======   =====================================
* (1..3:0)   Breakpoint     C Bp 3   line 93
                                     Main()
                                     node.f
```

See Also

context, status

QUIT QUIT

Terminate a debug session and exit IPD.

Syntax

quit

Description

The **quit** command terminates an IPD session. It is equivalent to the **exit** command. Either command terminates only those processes that the debugger has loaded.

Examples

1. Exit IPD:

```
(all:all) > quit
  *** IPD exiting...
```

See Also

exit

RECVQUEUE RECVQUEUE

Display pending receives.

Syntax

recvqueue [*context*] [*type*] [**-all**]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

type

The user-defined message as specified in the message-passing call. Only messages of the type specified by the *type* argument are displayed; otherwise, all message types in the context are displayed.

-all

Include both NX and MPI style messages in the display.

Description

The **recvqueue** command displays message receive requests that have been posted but not satisfied. Only receive requests posted by processes in the current or specified context are displayed. If a message *type* is specified then only receives for messages of that type on the nodes in the context are displayed.

RECVQUEUE *(cont.)*

RECVQUEUE *(cont.)*

The **-all** switch is useful for showing MPI messages for all communicators; without it, only MPI messages for the current context's communicator are displayed. The **-all** switch is also useful if an application uses both NX and MPI message passing, for example, if an MPI application is linked with a library using NX message passing. Once an application has executed an *MPI_Init*() call, only MPI-style receives are displayed by default, since in most cases the message passing occurring within the library is not a concern. However, since seeing these receives may be useful in understanding a problem, the **-all** switch causes NX receives to be displayed along with the MPI receives.

If the **msgstyle** command is set to NX, all receives (receives posted using MPI calls as well as those posted using NX calls) will be displayed as NX-style messages.

Processes that have unreceived messages posted are not necessarily blocked. The process may have posted one or more asynchronous receives (using, for example, **irecv()** or **hrecv()**) and continued executing. If the process has posted an **hrecv()** call, which requires a handler, the name of the handler is listed under the final column.

MPI applications that use the MPI_ANY_SOURCE or MPI_ANY_TAG will see the word "ANY" in the "For Msg From" and "Msg Tag" fields. Also, the "Call Type" and "Handler" fields are dropped when using the recvqueue command on MPI applications. Note that the "Msg Length" field is in number of bytes, not the number of elements, as was specified in the MPI receive function call.

In an MPI application, if the communicator used to post the outstanding receive has been freed the communicator name given in the display will be COMMUNKNOWN and the ranks listed will be relative to COMMWORLD.

Use the **msgqueue** command to display messages that have been sent but not received.

Examples

1. Display all receives that have not been satisfied by an incoming message:

```
(all:0) > recvq
 *** Unsatisfied receives posted in (all:0)
           Recv Posted    For Msg                  Msg Length
Call Type      By          From       Msg Type     (in bytes)      Handler
=========  ===========  ===========  ============ ============  ============
CRECV      (0:0)        (2:1)            100          8
```

RECVQUEUE *(cont.)* RECVQUEUE *(cont.)*

2. In an MPI application, display any receives that have not been satisfied by an incoming message within the COMMWORLD communicator:

```
(COMMWORLD:all) > recvq
 *** Unsatisfied receives posted in (COMMWORLD:all)
                                                          Msg Length
  Recv Posted By      For Msg From      Msg Tag      (in bytes)
================== ================= ============ ============
(COMMWORLD:10)      (COMMWORLD:0)        100              8
(COMMWORLD:23)      (COMMWORLD:ANY)        1             80
```

3. In an MPI application that is linked with the ProSolver library (which is written using NX message passing), display all receives that have not been satisfied by an incoming message. Receives posted by any communicator are included:

```
(COMMWORLD:all) > recvq -all
 *** Unsatisfied MPI receives posted in (COMMWORLD:all)
                                                          Msg Length
  Recv Posted By      For Msg From      Msg Tag      (in bytes)
================== ================= ============ ============
(COMMWORLD:10)     (COMMWORLD:0)         100             8
(COMMWORLD:23)     (COMMWORLD:ANY)         1            80
(COMM2:6)          (COMM2:ANY)           ANY            10

*** Unsatisfied NX receives posted in (all:0)
              Recv Posted       For Msg                        Msg Length
Call Type         By              From          Msg Type     (in bytes)
=========     ===========    ===========    ============ ============
CRECV         (0:0)            (2:1)           200021           8
```

See Also

msgqueue

REMOVE REMOVE

Remove breakpoints, watchpoints, and tracepoints.

Syntax

remove [*context*] [*actionpoint_number* [*actionpoint_number*] ...] | **-all**

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

actionpoint_number

The number of the breakpoint, watchpoint, or tracepoint to be removed. To determine the *actionpoint_number*, use the **break**, **watch**, or **trace** commands.

-all

The **-all** switch removes all breakpoints, watchpoints, and tracepoints in the default or specified context.

Description

The **remove** command removes the specified breakpoints, watchpoints or tracepoints, or all action points in the default or specified context.

You may remove nodes from an action point context by using the **remove** command with the desired nodes in the context argument. The IPD program does not remove the action point, but rather removes the nodes from the action point context. Only when all the nodes have been removed from the action point context is the action point removed.

REMOVE *(cont.)*

REMOVE *(cont.)*

When you remove an action point its number is no longer valid, but the number is not used again in the same debug session.

This command may not be used while examining core files.

Example

1. Display all current breakpoints, remove breakpoint 1 on (0:0), then redisplay the breakpoints:

```
(0:0) > break (all:0)

Bp #  File name   Procedure  Breakpoint Condition      Bp context
====  =========   =========  ====================      ==========
   1  gauss.f     shadow     Call shadow               (0:0)
   3  gauss.f     shadow     Line 175 after 10         (1..3:0)
   4  gauss.f     shadow     Line 180                  (all:0)
(0:0) > remove (0:0) 1
(0:0) > b (all:0)

Bp #  File name   Procedure  Breakpoint Condition      Bp context
====  =========   =========  ====================      ==========
   3  gauss.f     shadow     Line 175 after 10         (1..3:0)
   4  gauss.f     shadow     Line 180                  (all:0)
```

2. Remove breakpoint 3 on (2:0), then redisplay the breakpoints:

```
(0:0) > remove (2:0) 3
(0:0) > b (all:0)

Bp #  File name   Procedure  Breakpoint Condition      Bp context
====  =========   =========  ====================      ==========
   3  gauss.f     shadow     Line 175 after 10         (1,3:0)
   4  gauss.f     shadow     Line 180                  (all:0)
```

See Also

break, watch, trace

RERUN RERUN

Reload and restart the execution of the program, clearing previous command line arguments.

Syntax

rerun [<*infile*] [> *outfile*] [*program_args*]

Arguments

infile The program's input file argument. All of the program's input is redirected from *infile.*

outfile The program's output file argument. All of the program's output is redirected to *outfile.*

program_args Arguments to be passed to the program. Anything following the file-redirection arguments, up to the end of the line, a non-escaped pound sign, or a non-escaped semicolon, is used as a program argument. See the **load** command for more information on program arguments.

Description

The **rerun** command reloads and executes a program from its beginning without using command-line arguments from a previous **load, run** or **rerun**. All data in the program is re-initialized. (Use the **continue** command to continue execution of a stopped or breakpointed process without reloading the program or reinitializing data.)

To include the special characters ";", "#", "$", or "\" as arguments, they must be escaped (preceeded) with a backslash character ("\").

The **rerun** command does the following:

1. Kills the current program, which deletes all outstanding messages for the application.

2. Reloads the program.

3. Resets all of the user breakpoints and instrumentation if the program arguments have not changed from the last **load, run**, or **rerun** command.

4. Resets the argument list.

5. Starts executing the program.

RERUN *(cont.)* RERUN *(cont.)*

To restart the application without retyping the previous command line arguments, use the **run** command. The input redirection is not saved between **run** commands, so you need to respecify it if you issue another **run** command.

This command returns an error if it is used while examining a core file.

Examples

1. Load the *gauss* program on nodes 0..3 with program arguments **-d -f gauss**:

```
ipd > load gauss -on 0..3 -d -f gauss
*** reading symbol table for /home/myacct/gauss... 100%
*** loading program...
*** initializing IPD for parallel application...
*** load complete
```

Start the program and wait for it to complete:

```
(all:0) > continue; wait
*** interrupt...
```

After that completes, restart the program, this time without program arguments:

```
(all:0) > rerun -on 0..3
* This command will destroy all processes under debug.
  Are you sure you want to do this (y/n)? y
*** reading symbol table for /home/myacct/gauss... 100%
*** initializing IPD for parallel application...
```

See Also

run, continue, stop, wait, signal

RUN RUN

Reloads and restarts the execution of a program, reusing previous command line arguments.

Syntax

run [<*infile*] [> *outfile*] [*program_args*]

Arguments

infile The program's input file argument. If specified, all of the program's input is redirected from *infile.*

outfile The program's output file argument. All of the program's output is redirected to *outfile.*

program_args Arguments to be passed to the program. Anything following the file-redirection arguments, up to the end of the line, a non-escaped pound sign, or a non-escaped semicolon, is used as a program argument. Refer to the **load** command for more information about program arguments.

Description

If a program is in its initial state (just after it is loaded), **run** causes it to begin executing. If the program is in any other state, the **run** command reloads and executes a program from its beginning, retaining command-line arguments from a previous **load**, **run**, or **rerun** if no new arguments are specified. (Use the **continue** command to continue execution of a stopped or breakpointed process, or to run in a specified context.)

To include the special characters ";", "#", "$", or "\" as arguments, they must be escaped (preceeded) with a backslash character ("\").

If you assign a value to a variable, the **run** command resets it to the initial value. Use either the **continue** or the **step** command to retain the assigned value of a variable.

The **run** command does the following:

1. Kills the current program, which deletes all outstanding messages.

2. Reloads the program.

3. Resets all of the user breakpoints and instrumentation if the run command is used without arguments or if the arguments have not changed from the last **load**, **run**, or **rerun** command.

RUN *(cont.)* RUN *(cont.)*

4. Resets the argument list if program arguments are specified.

5. Starts executing the program.

The IPD program executes application processes asynchronously. The input redirection is not saved between **run** commands, so you need to respecify it if you issue another run command.

Use the **wait** command to wait for all processes in a context to stop.

A **run** command that does not specify any application command line arguments reuses the argument list from the last **run** or **rerun** command. To restart the application without using the previous command line arguments, use the **rerun** command.

See the description of the **load** command for more information on application command line arguments.

This command returns an error if it is used while examining a core file.

Examples

1. Load the program *gauss* with program arguments **-d -f gauss**.

```
ipd > load gauss -d -f gauss
*** reading symbol table for /home/myacct/gauss... 100%
*** loading program...
*** initializing IPD for parallel application...
*** load complete
```

Start the program and wait for it to complete.

```
(all:0) > continue ; wait
*** interrupt...
```

After that completes, restart the program using the same arguments.

```
(all:0) > run
```

See Also

rerun, continue, stop, wait, signal

SET SET

Set or display IPD variables.

Syntax

List all set variables:
set

List variable definition:
set *variable_name*

Define new or redefine old variable:
set *variable_name string*

Arguments

variable_name The symbolic name of the command line variable you are defining.

string The *string* argument includes all text after the *variable_name* to the end of the command line. To include pound signs, semicolons, non-substituting dollar signs, or backslashes as part of the string, escape (precede) them with a backslash ("\") character. You may build a command line variable from other command line variables by specifying a previously defined *variable_name* prefixed with a dollar sign ($) in the *string*. If the dollar sign is not escaped, substitution will occur when the **set** command is entered. If the dollar sign is escaped, substitution will occur when the variable being **set** is used. Escaping the dollar sign for variables in the definition of a new variable allows using variables in the definition that are not yet defined.

Description

The **set** command allows you to set or display command line variables. Command line variables are expanded immediately unless the dollar sign is escaped with a backslash. A recursive variable definition generates an error when you use it.

To use a command line variable in a command, precede the *variable_name* with a dollar sign ($). The *variable_name* must be followed by a space to separate it from the next argument on the command line. If you do not wish a space after the *variable_name*, enclose it in braces as follows:

${*variable_name***}**

SET *(cont.)* SET *(cont.)*

Use the **unset** command to delete command line variables. You may create an alias for the **unset** command, but you may not use "unset" as an alias.

You may not follow the **set** command with another command on the same command line.

Examples

1. Define the command line variable *myproc* as (1..3:0). Then, use this command line variable in the **context** command:

```
(0:0) > set myproc (1..3:0)
(0:0) > context $myproc
(1..3:0) >
```

2. Display the current command line variables:

```
(1..3:0) > set
  Variables   Substitution String
  =========   ===================
  myproc        (1..3:0)
```

3. Set *x* to the command line variable *long_name[104]*. Alias **an** to **assign** in the *$myproc* context. Use **an** to assign the variable *x* (that is, **an** becomes an alias for the command string **assign (1..3:0)** in this example):

```
(1..3:0) > set x long_name[104]
(1..3:0) > alias an assign $myproc
(1..3:0) > an $x = 100
```

See Also

alias, unalias, unset

SIGNAL SIGNAL

Set or display the signal-reporting mask.

Syntax

Display the current signal-reporting mask:
signal [*context*]

Enable signal reporting for specified signals:
signal [*context*] **-on** {*signo* [*signo*]... | **-all** }

Disable signal reporting for specified signals:
signal [*context*] **-off** {*signo* [*signo*]... | **-all** }

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

signo

The signal to be added or removed from the signal-reporting mask. The signal may be specified either as a number or a symbolic name (such as *SIGCHLD*). The symbolic names are defined in */usr/include/signal.h.*

-on

Add the specified signals to the signal-reporting mask. By default IPD reports all signals except SIGALRM and SIGCHLD.

-off

Remove the specified signals from the signal-reporting mask.

-all

Apply the specified command (**-on** or **-off**) to all signals.

SIGNAL *(cont.)* SIGNAL *(cont.)*

Description

IPD maintains a signal mask for each process under its control. The signal mask represents the set of Unix signals that are reported, and initially includes all signals except for SIGALRM and SIGCHLD. The **signal** command with no switch arguments displays the current signal mask for the default or specified context. The **-on** switch specifies that the list of signals is to be enabled in the mask, and the **-off** switch specifies that the signals are to be disabled.

NOTE

> The signal-reporting mask does not affect the action that the signal has on the application—it only affects whether IPD reports the receipt of the signal. For example, if a process receives a *SIGSEGV* signal, and that signal is not enabled in the signal-reporting mask, the process is still killed, but the next process state that IPD reports is “Exited”, rather than “Signaled”.

This command is useful when running applications that expect to receive certain signals. Using **signal -off** command tells IPD to ignore these signals, making it easier to run to the point of interest without having to continue the application for every signal that comes in.

The symbolic signal names recognized by the signal command are those defined in */usr/include/signal.h*:

```
SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGEMT,
SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGALRM,
SIGTERM, SIGURG, SIGSTOP, SIGTSTP, SIGCHLD, SIGTTIN, SIGTTOU,
SIGIO, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGWINCH, SIGINFO,
SIGUSR1, SIGMIGRATE
```

The Unix signals SIGUSR2 and SIGCONT are not accepted by the **signal** command—they are reserved.

This command may not be used while examining core files.

SIGNAL *(cont.)*

SIGNAL *(cont.)*

Examples

1. This example displays the current set of signals that will not be reported:

```
(all:0) > signal
  Signals not reported by IPD:
  ***** (all:0) *****
  SIGALRM SIGCHLD
```

2. This example specifies that IPD should not report receiving the SIGSEGV signal—if a process faults with this signal, it will simply exit:

```
(all:0) > signal -off SIGSEGV

(all:0) > signal
  Signals not reported by IPD:
  ***** (all:0) *****
  SIGALRM SIGCHLD SIGSEGV
```

See Also

continue, **run**, **rerun**, **wait**

SOURCE SOURCE

Set or display the current source directory search paths.

Syntax

Display source directory search path:
source [*filename*]

Set new source directory search path:
source *filename directory* [*directory*] ...

Add directories to source directory search path:
source [*filename*] **-add** *directory* [*directory*] ...

Remove directories from source directory search path:
source [*filename*] **-remove** *directory* [*directory*] ...

Arguments

filename	The name of a previously loaded executable file, used to specify which program's search path to access. If a file name is not specified, the command applies to all executable files
directory	A list of path names for the directories that contain the application source files.
-add	Add the specified directories to the source directory search path. The directories specified are appended to the end of the search path.
-remove	Remove the specified directories from the source directory search path.

Description

The **source** command with no arguments displays the search paths for all loaded modules. If you specify a filename, the search path for that file is displayed. When adding or deleting directories from the search paths, if the load module name is omitted the change is applied to the search paths for all load modules.

The directories are listed in the order that IPD uses to search for a source file for the **list** command. The default directory search path assigned at load time is the current directory (.). A directory must exist and be readable to be added to the search list. If a non-existent directory is specified in a list of directories to be added, an error message is displayed, and only the directories that precede the non-existent directory in the list are added.

SOURCE *(cont.)* SOURCE *(cont.)*

Examples

1. Display the current source directory search path for the previously loaded program, *gauss*. Add */usr/you/Fpi/* to the source directory search path and list the node program:

```
(all:0) > source gauss
  Source search paths for gauss:
        .
(all:0) > source gauss -add /usr/you/Fpi
(all:0) > source gauss
   Source search paths for gauss:
        .
        /usr/you/Fpi
(all:0) > list,10
***** (all:0) ***** gauss.f
57      program gauss
58
59      include ’nx.h’
60
61     integer SIZETYPE, INITTYPE, PARTTYPE, MSGSIZE, CUBESIZE,
62     >                  HOST, HOSTPID, APPLPID, DOUBLESIZE
63
64      integer*4 worknodes, mynode, pid, size
65      integer*4 basicpoints, extrapoints, mypoints, i, j
66      integer*4 starttime, points
(all:0) >
```

See Also

list, status

STATUS

STATUS

Display the debug environment settings and system partition information.

Syntax

status

Description

The **status** command displays version number, debug mode (runtime or core analysis), debug partition information (only if doing core file analysis), application partition information (only if one has been indicated via **load** or **coreload**), the state of the **more** command (the default is "on"), the state of the **threads** command (the default is "off"), the state of the **msgstyle** command (either NX or MPI), the name of the log file (if any) to which the output from the debug session is being written, and the source search paths for each executable under debug.

Examples

1. Display current status after loading an executable:

```
(all:0) > status

IPD version number: Paragon Release 1.4

Debug mode: Runtime process analysis
Application partition info: .compute.karla
USER    GROUP    ACCESS    SIZE    FREE    RQ    EPL
karla   tools      777        6       0    SPS     5

Message style: NX
More: on
Threads: off
Log file: /home/karla/debug.log

Source search paths for /home/karla/tests/apps/myapp:
.
```

STATUS *(cont.)*

STATUS *(cont.)*

2. Display status after loading core files from an NX application for analysis:

```
(0:0) > status

IPD version number: Paragon Release 1.4

Debug mode: Core file analysis
Debug partition name: karla
Debug partition size: 1
Core directory path: /home/karla/tests/apps/core

Number of nodes in application: 4
Application partition info:
USER    GROUP    ACCESS    SIZE    FREE    RQ    EPL
karla   tools      777        2       0    SPS     5

Message style: NX
More: on
Threads: off
Log file:

Source search paths for /home/karla/tests/apps/myapp:
.
```

3. Display status after loading an MPI application core file:

```
(host) > status

IPD version number: Paragon Release 1.4

Debug mode: Core file analysis
Debug partition name: karla
Debug partition size: 1
Core directory path: /home/karla/tests/fault/core

Unix application core file.

Message style: MPI
More: on
Threads: off
Log file:

Source search paths for /home/karla/tests/fault/segfault:
.
```

See Also

process, more, log, load, coreload

STEP STEP

Single step through the processes in the current or specified debug context.

Syntax

Step through source line(s):
step [*context*] [**-call**] [,*count*]

Step one machine instruction:
step [*context*] **-instruction** [**-call**] [,*count*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-call Treat all subroutine and function calls as single statements. If **-call** is not specified, routines compiled with line-number information are entered and their statements stepped through.

-instruction Step one instruction instead of stepping one source line.

count The number of source lines or instructions to step through before returning control to the user. The default *count* is one source line or machine instruction.

Description

The **step** command executes a program one source line or one machine instruction at a time. Upon returning control to the user from a **step** command, IPD displays process information with the **process** command.

STEP *(cont.)* STEP *(cont.)*

When stepping through source line numbers, any procedures compiled without line number information are treated as if the command were **step -call**, even if you did not specify **-call**.

When stepping through machine instructions, you cannot step through a system call trap instruction to the operating system. The trap is treated as if the command were **step -instruction -call.**

Single-stepping is synchronous; the **step** command does not return until all processes in its context have stepped. If your program blocks during the **step** command, use the interrupt signal (pressing `<Del>` or `<Ctrl-C>`) to regain the IPD prompt. At this point the current state of the process is still "Executing". Use the **stop** command to stop the process.

This command may not be used while examing core files.

Examples

1. Assume that the program is stopped at line 93 on all nodes, just before a **crecv**(). Step to line 94:

```
(all:0) > process
   Context    State          Reason        Location   Procedure
  =========   ======         ======        ========   =========
  (all:0)     Breakpoint     C Bp 3        line 93    Main()

(all:0) > list 5
***** (all:0) ***** node.f
93      call crecv(SIZETYPE, size, PARTSIZE)
94      worknodes = size
95
96c
97c     receive integration parameters

(all:0) > step
   Context    State          Reason        Location   Procedure
  =========   ======         ======        ========   =========
* (all:0)     Stepped                      line 94    Main()

(all:0) >
```

See Also

continue, break, trace, watch

STOP STOP

Stop program execution in the current context.

Syntax

stop [*context*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

Description

The **stop** command stops program execution. Processes that are blocked waiting for something, such as a **crecv()**, are not in a stopped state, but are still executing. Stopping program execution when you do not have an IPD prompt requires that you send an interrupt signal (entering `<Del>` or `<Ctrl-C>`) so you can get a prompt at which you can enter a **stop** command to stop application processes. Many IPD commands require processes to be stopped so that valid information can be obtained from the operating system.

This command may not be used while examining core files.

STOP *(cont.)* STOP *(cont.)*

Examples

1. A program named *gauss* blocks at its first receive. Send an interrupt signal, then issue the **process** command. This indicates the program is still executing. Issue the **stop** command, and then the **process** command again:

```
(all:0) > run; wait
 *** interrupt...
(all:0) > process
 Context      State        Reason    Location        Procedure
 =======      =====        ======    ========        =========
*(all:0)      Executing
(all:0) > stop
(all:0) > p
 Context      State        Reason    Location        Procedure
 =======      =====        ======    ========        =========
*(all:0)      Interrupted             0x00016648      _mcmsg_flick
```

See Also

process, kill

SYSTEM SYSTEM

Execute a shell command.

Syntax

system *shell_command*

or

! *shell_command*

Arguments

shell_command A string consisting of operating system shell commands (not an IPD command) to be executed. All text following the **!** or **system** to the end of the line, a non-escaped semi-colon, or a non-escaped pound sign is part of the *shell_command* argument. To include any semi-colons, pound signs, dollar signs, or backslashes, escape (preceed) them with a backslash character.

Description

Use either the **system** or **!** command to execute an operating system shell command from within the IPD program. The *shell_command* argument is not interpreted by the IPD program. All *shell_command* text to the end of the **system** command line or to a non-escaped semi-colon character or a non-escaped pound sign is passed directly to **sh** (the Bourne shell). All variables that begin with a non-escaped dollar sign are expanded before being passed to the shell. You may not follow the **system** command with any other commands on the same command line.

If a log file is active, output from this command is written in the log file.

Examples

1. Issue the shell command **ls -l** from within the IPD program:

```
(all:0) > system ls -l /usr/paragon/examples/fortran/gauss
total 23
-r--r--r--   1 root    other       1413 Mar 30 21:03 README
-r--r--r--   1 root    other        187 Mar 30 21:03 gauss.f
-r--r--r--   1 root    other        475 Mar 30 21:03 makefile
(all:0)>
```

THREADS THREADS

Controls number of threads displayed for each process with the **display**, **frame**, and **process** commands.

Syntax

threads [**-off** | **-on**]

Arguments

-off Display information for a single thread. The thread reported on will be the thread that caused the process to stop. This is the default behavior.

-on Display information for all user threads in each process.

Description

The **threads** command allows you to specify whether information about more than one thread should be reported by the **display**, **frame**, and **process** commands.

Use the **threads** command without any arguments to display its current setting.

An application compiled with **-nx** has a minimum of three user threads, the main user thread, an *hrecv* thread (which remains idle unless one of the hrecv handler functions is called), and a message-passing paging thread (*VM pager*). If **-Mconcur** is added to the compile line, an additional thread is created for executing loops in parallel.

An application compiled with **-lnx** starts off with a single user thread. The *VM pager* and *hrecv* threads are created if a call to *setptype()* is executed.

Applications that do their own thread management via explicit calls to the pthreads library (instead of using **-Mconcur)** may have additional threads that result from executing calls that create and delete threads.

By default (**threads** is set to "off"), all commands report on a single thread, which is usually the main user thread. If a different thread stops for any reason (a fault or a breakpoint), then information concerning that thread is reported instead.

THREADS *(cont.)* THREADS *(cont.)*

When displaying threads, a thread ID number is associated with each thread displayed. This is a number assigned by the debugger, starting at 0, for each thread being displayed. It can only be used to relate thread output from various commands (a trace back and register display) while the application is stopped. Once execution is resumed, it is possible for a thread to have a different ID associated with it when the process is next stopped and thread information is displayed again.

The main user thread always has ID 0 throughout an application's execution. An application compiled with **-lnx** only has thread 0. An application compiled with **-nx** has the main user thread ID 0, an *hrecv* thread ID 1, and a message-passing paging thread ID2. If **-Mconcur** is used then the ID for that thread is 3. If pthread creation calls are used in place of **-Mconcur**, thread ID 3 will be a thread created by one of these calls and there may be additional threads in the display.

The general rule for a thread ID to be constant throughout the execution of an application is that a thread and all threads created prior to it be created once and live throughout the life of the application. When an application does its own pthreads management, or in any way changes the order in which implicit thread creation takes place, no assumptions can be made about a certain thread ID representing a specific thread. Stack tracebacks should be used to determine which thread is which in these cases. The **frame** command is useful for determining the origin of the threads in a display.

Examples

1. Display information on all user threads when using the **display**, **frame**, and **process** commands:

```
(all:0) > threads -on
```

2. Display the current thread display state:

```
(all:0) > threads
 Threads: on
```

TRACE TRACE

Set a tracepoint or display current tracepoints.

Syntax

Display Tracepoint information:
trace [*context*] [**-full**]

Set Tracepoint at procedure:
trace [*context*] [*file*{}] *procedure*() [**-after** *count*]

Set Tracepoint at source line number:
trace [*context*] [*file*{}] [*procedure*()] #*line* [**-after** *count*]

Set Tracepoint at instruction address:
trace [*context*] *address* [**-after** *count*]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

-full

Displays tracepoint information in a long or "full" format, with more room for file, class, and procedure names.

file

The name of the source module in which the procedure or line resides. To refer to a file other than the location of the current execution point, you must prefix the line number with *file*. When you refer to a procedure, you can omit the *file* name unless there are duplicate procedure names, because the IPD program can find the source file from the symbol table information.

TRACE *(cont.)* TRACE *(cont.)*

procedure

The optional *procedure* argument is the name of the procedure at which you want to set the tracepoint, or the procedure in which the line number you are specifying resides. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class::

The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::...*.

type

Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

line

The source line number at which you want to set the tracepoint. The line number must be preceded with a pound sign (#). In general, the statement must be executable. For example, you cannot set a tracepoint on a Fortran **FORMAT** statement, a comment, or an empty line. The trace message is displayed just before executing the specified statement. To qualify the line number, use the *file* and/or *procedure* qualifiers.

address

The address must be an instruction address (not a data address). The trace message is displayed just before executing the instruction at the *address*.

TRACE *(cont.)* TRACE *(cont.)*

-after *count* In all forms of the **trace** command, the *count* argument is a positive integer indicating the number of times this tracepoint is encountered before the trace message is displayed. The default count is 1. For example, if you have a Fortran loop defined by the following

```
DO 10  I = 1,100
```

and you want to see a trace message after every fifth iteration, you would set the tracepoint inside the loop with an **-after** *count* of 5.

Description

Tracepoints are breakpoints that cause a message to be displayed rather than execution to halt. They can be used in place of inserting print statements in the source code to determine the execution path of the code. The following is a sample trace message:

```
(0:0) TRACEPOINT #1: gauss.f{}shadow()#150
```

Without any arguments the **trace** command lists all tracepoints whose context has any node or process in the current default context or *context* argument. An example of the **trace** command display is as follows:

```
(all:0)
Tp #  File name   Procedure  Tracepoint Condition        Tp context
====  =========   =========  ====================        ==========
   1  gauss.f      shadow     Line 150                   (all:0)
```

In the preceding display, the first line shows the current context for the **trace** command. The labeled columns denote the following:

Tp # The number of each tracepoint. The tracepoint number is used as an argument to the **remove** command.

File name The name of the source file associated with the tracepoint.

Procedure The name of the procedure where the code is located.

Tracepoint Condition
The condition under which the tracepoint will occur. The **-after** clause is not displayed unless the *count* is greater than 1.

TRACE *(cont.)* TRACE *(cont.)*

Tp context The tracepoint context. If the text overflows the "File name", "Procedure" and "Tracepoint Condition" columns, the right-most characters of the text are truncated. However, if the context overflows the "Tp context" field, the display for the tracepoint is continued on the next line. This is denoted by blanks in all fields except the "Tp context field", which contains the continued tracepoint context.

In some cases, the file and procedure names may be long enough that truncating them is not an option. In that case, you may use the **-full** switch to use an expanded format for the tracepoint display. The expanded format includes separate lines for the file name, procedure name, and tracepoint condition.

Breakpoints and tracepoints are not allowed at the same location at the same time. An error message is displayed if this is attempted.

If a single C statement consists of multiple source lines, set the breakpoint at the ending line. If a single C++ statement consists of multiple source lines, set the breakpoint at the starting line. For a multiple-line Fortran statement, set the breakpoint on the first line.

When you set a tracepoint on a function such as the following:

```
trace my_function()
```

The tracepoint is set on the first line of the function, if the function was compiled with symbols. If it was not compiled with symbols, or line number information has been stripped, the tracepoint is set on the function's entry point. As a result, if you set a tracepoint on a function, and then attempt to set a breakpoint on the first executable line of the same function, you will get a "tracepoint already exists" error.

This command may not be used while examining core files.

Examples

1. Set a tracepoint at the procedure **shadow()** in the current source file for node 0, process type 0 only:

```
(0:0) > trace shadow()
```

2. Set a tracepoint at line number 175 in the file *gauss.f*. Set the tracepoint so that the trace occurs at the beginning of the tenth execution of the function for process type 0 on nodes 1, 2, and 3:

```
(all:0) > trace (1..3:0) gauss.f{}#175 -after 10
```

TRACE *(cont.)*

TRACE *(cont.)*

3. Set a tracepoint at line number 180 in the source file *gauss.f*:

   ```
   (all:0) > trace gauss.f{}#180
   ```

4. Set a tracepoint at the start of a C++ member function **row** for the class **board**. Specify the argument types for the member function to distinguish between the desired function and another member function with the same name:

   ```
   (all:0) > trace board.c{}board::row(int)
   ```

5. Display the current tracepoints. The **trace** command displays those tracepoints that have a node or process type in the current context. The display context is shown on the line before the table and the tracepoint context is shown in the right most column of the display:

```
(0:0) > trace (all:0)

Tp #  File name   Procedure  Tracepoint Condition         Tp context
====  =========   =========  ====================         ==========
   1  gauss.f     shadow     Call shadow                  (0:0)
   3  gauss.f     shadow     Line 175 after 10            (1..3:0)
   4  gauss.f     shadow     Line 180                     (all:0)
```

See Also

break, watch, step

TYPE TYPE

Display the type of variables in the current or specified context.

Syntax

Display type of variable in current scope of context:
type [*context*] *variable*

Display type of global or static C variable:
type [*context*] *file*{} *variable*

Display type of local procedure variable:
type [*context*] [*file*{}] *procedure*() *variable*

Display type of a variable local to a block (in C or C++):
type [*context*] [*file*{}] *#line variable*

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

variable The *variable* argument is the symbolic name of the variable for which information is to be displayed. Alternatively, any expression that can appear on the left side of an assignment may be used in place of a simple variable name.

TYPE *(cont.)*

TYPE *(cont.)*

For C, C++, or Fortran programs, IPD follows the scoping rules of the language in use. For assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information and IPD will use C scoping rules. IPD looks for variables in the following places, in order:

- In the current code block.
- In the current procedure.
- For C++, IPD searches next for class member variables.
- In the static variables local to the current file.
- In the global program variables.

To specify variables not in the current scope, prefix the variable name with the *file{}*, *procedure()* and/or *#line* qualifiers. C++ class member variables may also be prefaced with the class name, as follows:

```
[[class]::[class::]...]variable
```

Use language-specific syntax to specify a variable. For example, in Fortran you would specify an element of a two-dimensional array as **a(1,1)**; in C or C++, it would be **a[1][1]**

file The name of the source module in which the variable resides. To refer to a file other than the location of the current execution point, you must prefix the variable name with *file*. When you refer to a procedure, you can omit the *file* name unless there are duplicate procedure names, because the IPD program can find the source file from the symbol table information.

procedure The optional *procedure* argument is the name of the procedure in which the variable resides. You need to specify the procedure when the execution point is not in the same procedure as the variable. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

TYPE *(cont.)* TYPE *(cont.)*

class:: The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

type Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

#line A line number from which the variable that you are specifying is accessible. You only need to specify a line number if the variable you are interested in is hidden by another variable of the same name in the current scope. Specifying any line number from which the desired variable is accessible allows IPD to find the variable.

Description

The **type** command shows the type of a specified variable. If a variable has a structured type, such as a C "struct" or "union" or a C++ "class", IPD displays the type information for the members of the type.

The **type** command displays the type of a variable. The scope of the thread that was active when the process stopped is used to qualify the variable. If the type of the variable within a different scope is needed, the variable must be qualified on the command line with a routine name and/or a file name.

Examples

1. Determine the type of the Fortran variable *tms* in process type 0 on node 0:

```
(all:0) > type (0:0) tms
  ** tst.f{}main()#5 tms **
   ***** (0:0) *****
       INTEGER
(all:0) >
```

TYPE *(cont.)* TYPE *(cont.)*

2. Determine the type of a C structure variable *msg*:

```
(all:0) > type msg
  ** tst.c{}main()#8 msg **
   ***** (all:0) *****
      struct msg_type {
           double a;
           double b;
           int points;
      };
(all:0) >
```

3. Determine the type of the C++ class member variable *lengths* in the class **rectangle**:

```
(all:0) > type rectangle::lengths
  ** shapes.C{}main(void)#36 rectangle::lengths **
   ***** (all:0) *****
      int *;
(all:0) >
```

4. Determine the type of variable *a_expr* which is a C++ class object:

```
(all:0) > type a_expr
  ** expression.C{}main(void)#45 a_expr **
   ***** (all:0) *****
      class expr {
           int row;
           int col;
           double f1;
           double f2;
           double f3;
      };
(all:0) >
```

See Also

display

UNALIAS UNALIAS

Delete previously defined aliases.

Syntax

unalias {*alias_name* [*alias_name* ...] | **-all**}

Arguments

alias_name	A string that was chosen as an alias for an IPD command using the **alias** command.
-all	Remove all currently defined aliases.

Description

The **unalias** command removes a previously-defined alias. Use the **alias** command without arguments to display the current list of alias names. You can create an alias for the **unalias** command, but you cannot use the name "unalias" as an alias.

Examples

1. Remove the alias **ct**:

```
(all:0) > alias
   Alias     Command String
    ======    ==============
    ct        context
(all:0) > unalias ct
(all:0) > alias
    Alias     Command String
     ======    ==============
(all:0) >
```

See Also

alias, set, unset

UNSET

UNSET

Delete previously defined command line variables.

Syntax

unset {*variable_name* [*variable_name*] ...] | **-all** }

Arguments

variable_name The symbolic name of the command line variable you are deleting. *Do not* precede the *variable_name* to be unset with a **$**.

-all Remove all currently defined command line variables.

Description

The **unset** command removes the definitions of command line variables previously defined with the **set** command. Use the **set** command with no arguments to display a list of the current command line variable names. You can create an alias for the **unset** command, but you cannot use "unset" as an alias.

Examples

1. Delete the command line variable *myproc*.

```
(0:0) > set
        Variables   Variable String
        =========   ===============
        myproc        (1..3:0)
(0:0) > unset myproc
(0:0) > set
         Variables   Variable String
         =========   ===============
(0:0) >
```

See Also

set, alias, unset

WAIT WAIT

Wait until all processes within the context have stopped running.

Syntax

wait [*context*]

Arguments

context

The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(**host**)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

Description

The **wait** command causes IPD to return with the prompt only when *all* processes within the context are in a "stopped" state (use the **process** command for process state information).

A program's output written to *stdout* appears between IPD commands and is not intermixed with IPD output. If the program needs to read from the terminal, you must use the **wait** command to process the read requests. To redirect the program's standard input, use the redirect argument in the **load**, **run**, or **rerun** command.

After a **run**, **rerun**, or **continue** command, the IPD program immediately issues a prompt. To cause IPD to withhold the prompt until a process hits a breakpoint or terminates, use the **wait** command.

Upon returning control to the user from **wait**, the IPD program uses the **process** command to display the process information.

WAIT *(cont.)* WAIT *(cont.)*

After you have issued a **wait**, if you decide not to wait for all the processes to stop running, use the interrupt signal (pressing **<Del>** or **<Ctrl-C>**) to regain the IPD prompt.

This command may not be used while examining core files.

Examples

1. Issue a **run** command followed by a **wait**. When all the processes have stopped running, the **wait** command issues a **process** command, and then returns a prompt:

```
(all:0) > run ; wait
 Context        State        Reason    Location        Procedure
 =======        =====        ======    ========        =========
*(all:0)        Breakpoint   C Bp 1    Line 150        shadow()
(all:0) >
```

See Also

continue, **run**, **rerun**, **stop**

WATCH WATCH

Set a watchpoint (data breakpoint) or display current watchpoints.

Syntax

Display watchpoint information:
watch [*context*] [**-full**]

Set watchpoint on an expression:
watch [*context*] [**-access** | **-write**] [*file*{}] [*procedure*()] *expression* [*#line*] [**-after** *count*]

Set watchpoint on an expression containing variables in the current scope of context:
watch [*context*] [**-access** | **-write**] *expression* [,*count*]

Set a watchpoint on an expression containing global or static C variables:
watch [*context*] [**-access** | **-write**] *file*{}*expression* [,*count*]

Set a watchpoint on an expression containing a local procedure variable:
watch [*context*] [**-access** | **-write**] [*file*{}]*procedure*()*expression* [,*count*]

Set a watchpoint on an expression containing variables local to a block in C or C++:
watch [*context*] [**-access** | **-write**] [*file*{}] *#line expression* [,*count*]

Set watchpoint on a memory address:
watch [*context*] [**-access** | **-write**] *address* [**-after** *count*]

Arguments

context The *context* argument specifies the context as a list of processes using either NX or MPI process naming conventions. An NX process consists of a node number and ptype. An MPI process consists of a communicator and rank. The node number, ptype, and rank may be expressed as a single value, a comma-separated list, a range, or a combination thereof. The keyword **all** may be used in place of any of these values as well. The special value **host** may be used in lieu of a process name to specify the controlling process(es) running in the service partition.

(host)
(**host** : {**all** | *ptypelist*})
({**all** | *nodelist*} : {**all** | *ptypelist*})
(*communicator* : {**all** | *ranklist*})

For more information, see the **context** command.

WATCH *(cont.)* WATCH *(cont.)*

-full Displays watchpoint information in a long or "full" format with more room for file, class and procedure names.

-access Specifies that a break will occur when the program accesses the specified variable or memory address (the break occurs just before the access). An access is either a read or a write. Break on access is the default if neither **-access** nor **-write** is specified. Use either the **process** or **wait** command to determine where in your source code the access occurred that caused the break.

-write Specifies that a break will occur when the program writes the specified variable or memory address (the break occurs just before the write). Use either the **process** or **wait** command to determine where in the source code the break occurred.

file The name of the source module in which a variable resides. To refer to a file other than the location of the current execution point, you must prefix the variable name with *file*. When you specify a procedure, you can omit the *file* name unless there are duplicate procedure names, because the IPD program can find the source file from the symbol table information.

procedure The *procedure* argument is the name of the procedure in which the variable resides. You need to specify the procedure when the execution point is not in the same procedure as the variable. The *procedure* argument must end with a pair of parentheses (()).

For C++, procedure names may include operator functions, such as **operator+()**, **operator new()**, or **operator int *()**. The operator names must include the "operator" keyword. You may also preface the procedure name with class information and may include argument types to distinguish between overloaded functions. The syntax is:

```
[[class]::[class::]...]procedure([type[,type]...])
```

class:: The name of the C++ class in which a procedure is a member function. Use the "**::**" without a class name to refer to a global procedure that is hidden by a member function in the current scope. Specify nested classes as *class1::class2::....*

WATCH *(cont.)* WATCH *(cont.)*

type Any legal C++ type specification, such as *int*, *float **, or *char (*)()*. Argument types may be omitted unless the procedure name is overloaded. For overloaded procedure names, you need only to specify enough arguments to uniquely identify the intended procedure. An error is reported when then procedure name is ambiguous.

#line A line number from which the variable that you are specifying is accessible. You only need to specify a line number if the variable you are interested in is hidden by another variable of the same name in the current scope. Specifying any line number from which the desired variable is accessible allows IPD to find the variable.

variable The required *variable* argument is the symbolic name of the variable upon which you want to set a watchpoint.

For C, C++, or Fortran programs, IPD follows the scoping rules of the language in use. For assembly language programs, you can use symbolic names if you have used the proper assembler directives to produce the symbolic debug information and IPD will use C scoping rules. IPD looks for variables in the following places, in order:

- In the current code block.
- In the current procedure.
- For C++, IPD searches next for class member variables.
- In the static variables local to the current file.
- In the global program variables.

To specify variables not in the current scope, prefix the variable name with the *file{}*, *procedure()* and/or *#line* qualifiers. C++ class member variables may also be prefaced with the class name, as follows:

```
[[class]::[class::]...]variable
```

Use language-specific syntax to specify a variable. For example, in Fortran you would specify an element of a two-dimensional array as **a(1,1)**; in C or C++, it would be **a[1][1]**.

WATCH *(cont.)*

WATCH *(cont.)*

-after *count* A positive integer indicating the number of times this watchpoint is encountered before execution is halted. The default count is 1. For example, if you have a Fortran loop defined by the following

```
DO 10  I = 1,100
```

and you set a watchpoint on *I* with an **-after** *count* of 5, the program will stop on every fifth iteration of the symbol *I*..

address The address to watch. The address must be a data address (use the **break** command to set a breakpoint on a code address). Watchpoints set on memory addresses cause a break to occur just before the memory access.

Description

The **watch** command sets a watchpoint on a specified variable. A watchpoint stops execution of an application upon reading or writing the variable.

Without any arguments the **watch** command lists all watchpoints whose context has any node or process in the current default context or *context* argument. An example of the **watch** command display is as follows:

```
(all:0)
Wp #  File name   Procedure  Watchpoint Condition         Wp context
====  =========   =========  ====================         ==========
   1  gauss.f      shadow     Write int_var                 (all:0)
```

In the preceding display, the first line shows the current context for the **watch** command. The labeled columns denote the following:

Wp # The number of each watchpoint. The watchpoint number is used as an argument to the **remove** command.

File name The name of the source file associated with the watchpoint. For global variables, the file name is set to "<global>".

Procedure The name of the procedure where the code or variable is located. For global or static variables the **Procedure** field is set to "<global>" or "<static>".

Watchpoint Condition
The condition under which the watchpoint will occur. The **after** clause is not displayed unless the *count* is greater than 1.

WATCH *(cont.)*

WATCH *(cont.)*

Wp context — The watchpoint context. If the text overflows the "File name", "Procedure" and "Watchpoint Condition" columns, the right-most characters of the text are truncated. However, if the context overflows the "Wp context" field, the display for the watchpoint is continued on the next line. This is denoted by blanks in all fields except the "Wp context" field, which contains the continued watchpoint context.

A single watchpoint is allowed per process.

When setting a watchpoint on any non-stack variable in a multi-threaded application (compiled with **-Mconcur**), the current execution point of the thread that caused the process to stop is used to resolve the address of the variable. The watchpoint is then set for all threads. If any thread accesses this address, execution is interrupted. In the case of local (stack) variables, the current execution point of the thread that caused the process to stop is used to qualify the variable. The address for the variable is determined and set for each thread. Each thread is watching a different stack address but you are watching a single variable that has copies on multiple threads. This makes it possible to watch loop variables that have been dispersed among several threads (by **-Mconcur** for instance). Specify an address instead of a variable name to override this feature.

In some cases, the file and procedure names may be long enough that truncating them is not a good option. In that case, you may use the **-full** switch to specify a expanded display format for the watchpoint display. The expanded format includes separate lines for the file name, procedure name and watchpoint condition:

```
  (all:0)

     File name
     Procedure
Wp # Watchpoint Condition                              Wp context
==== ==========================================    ==========
   1 gauss.f                                         (all:0)
     shadow
     Write int_var
```

Use the **continue** command to resume executing the application after a watchpoint.

This command may not be used while examining core files.

WATCH *(cont.)* WATCH *(cont.)*

Examples

1. Set a watchpoint on write to the address *0x0401b7a8* for nodes 0 and 1, process type 0:

```
(all:0) > watch (0,1:0) -write 0x0401b7a8
```

2. Set a data watchpoint when the variable *p1* is accessed for reading or writing on node 2:

```
(all:0) > watch (2:0) p1
```

3. Set a data watchpoint when the variable *row* in the C++ member function **position** is accessed for reading or writing. Note that you don't need to specify the class name if the scope of the current instruction pointer is in any member function for the class **board**.

```
(all:0) > watch (all:0) board::position()row
```

4. Set a data watchpoint when the C++ global variable *size* is accessed for reading on node 0. The current instruction pointer is in a member function and the global variable *size* is hidden by a class variable with the same name.

```
(all:0) > watch (0:0) ::size
```

5. Display the watchpoints for all processes. The **watch** command displays a watchpoint if its node and process type pair is in the current context. The display context is shown on the line before the table and the watchpoint context is shown in the right most column of the display:

```
(all:0) > watch

(all:0)
Wp #  File name  Procedure  Watchpoint Condition        Wp context
====  =========  =========  ====================        ==========
  1   myhello.c  main        Write 0x401b7a8             (0,1:0)
  2   myhello.c  main        Access p1                   (2:0)
```

See Also

trace, break, step

Using IPD With Host/Node Models A

Debugging Host/Node Programs on Paragon™ Systems

A host/node program is one that is written such that part of it (the "parent" or "controlling" process) runs in the service partition and that process starts processes running in the compute partition, or other processes running in the service partition (the "child" processes). The service partition is the "host"—it has the parent process running on it and may also have child processes running on it. The "node" child processes run on nodes in the compute partition.

Example

In the following example, a parent (host) program forks two child processes, one on the same node as itself and one on a node in the compute partition. The child forked onto the parent node is referred to as the host-child process, since it is running in the service partition. The other process is forked onto node 0 and is referred to as the node-child process. Both child processes are created with a ptype of 10. The parent process does not set its own ptype and therefore has none. Thus, the IPD context for each of these processes is:

(host)	Parent process
(host:10)	Child process on same node a parent
(0:10)	Child process on node 0

The parent process prints a *hello* message and then waits for the children to complete. The children each execute their own program to print a message identifying them.

Here is an IPD debug session illustrating how to swap between the processes to get each of them to run to completion. The complete code used in the example is included at the end of this section.

First load the parent program and then run it and wait for the child processes to be created. An information message is printed as the new processes are created.

When all of the processes are created, press **<Return>** to get the debugger prompt. If you had entered **run;wait** rather than just **run**, a **<Ctrl-C>** would be required to get a prompt.

```
ipd > load parent
 *** reading symbol table for /home/cjd/host/parent... 100%
 *** loading program...
 *** load complete
(host) > run
(host) >  *** initializing IPD for parallel application...

*** INFO: processes (host:10) have been created and are stopped.
*** INFO: processes (0:10) have been created and are stopped.

Hello from Parent (pid 131572)
(host) >
```

Next, the context is changed to include all ptypes running on the host node. The process (host:10) is the host-child process. The process (host: -131572) is the parent process. Since it did not give itself a ptype, the debugger assigns it a ptype of the negative value of its *pid* so that it can distinguish it from other host processes. Normally, this negative ptype is not displayed, since the program does not really have a ptype. When there are multiple host processes in a context, this negative ptype is displayed in order to distinguish between the processes.

```
(host) > context(host:all)
(host:-131572,10) >
```

The **process** command shows the state of the host processes. The parent process is in an "Executing" state because it is busy waiting for the child processes to complete. The host child is in the "Initial" state because it has been created but not run. IPD automatically stops a child process upon creation (via *fork(2)* or *exec(2)*) so that its execution can be monitored and controlled.

```
(host:-131572,10) > process
 Context           State       Reason    Location      Procedure
 =======           =====       ======    ========      =========
*(host)            Executing
*(host:10)         Initial               Line 28       main()
(host:-131572,10) >
```

To see what is happening with the node-child process, we enter another **process** command. This process is at the same point as the host child.

```
(host:-131572,10) > process(0:10)
 Context           State       Reason    Location      Procedure
 =======           =====       ======    ========      =========
*(0:10)            Initial               Line 28       main()
(host:-131572,10) >
```

We change the context to include just the host child and then begin its execution by using the **continue** command. The **wait** command is used in conjunction with the **continue** so that we know when the process is stopped. The process will be stopped by the debugger when it executes a new file via an *exec(2)* call. If the context had not been changed to exclude the parent process, a **<Ctrl-C>** would have been necessary to return to the prompt. While the **continue** has no effect on this process, because it is already running, the **wait** would not return until all processes in the context are stopped. The parent will not reach a stopped state until it returns from the *nx_waitall()*. Thus, the **<Ctrl-C>** would be needed to escape from the **wait** command.

```
(host:-131572,10) > context(host:10)
(host:10) > continue;wait
 Child forked...
 *** reading symbol table for /home/cjd/host/child_host... 100%

*** INFO: processes (host:10) have been created and are stopped.
 Context             State        Reason     Location       Procedure
 =======             =====        ======     ========       =========
*(host:10)           Initial                 Line 5         main()
(host:10) >
```

To execute the node-child process, we do the same as with the host-child process. The host- and node-child processes are started using separate commands because we cannot form a context node list that includes the keyword "host" and a node number. Refer to the **context** command for information about how to include a host node in a context with a compute node. For purposes of this example, we choose the clarity of having the distinct contexts.

```
(host:10) > context(0:10)
(0:10) > continue;wait
 Child forked...
 *** reading symbol table for /home/cjd/host/child_node... 100%

*** INFO: processes (0:10) have been created and are stopped.
 Context             State        Reason     Location       Procedure
 =======             =====        ======     ========       =========
*(0:10)              Initial                 Line 5         main()
(0:10) >
```

Now we continue the node-child process and then the host-child process, allowing them to run until completion. They will each print a message and then exit.

```
(0:10) > continue;wait
Hello from Node Child process (0:10)
 Context            State        Reason     Location       Procedure
 =======            =====        ======     ========       =========
*(0:10)             Exiting                 0x0001f840     __exit()
(0:10) > context(host:10)
(host:10) > continue;wait
Hello from Host Child process (2:10)
 Context            State        Reason     Location       Procedure
 =======            =====        ======     ========       =========
*(host:10)          Exiting                 0x0001f840     __exit()
(host:10) >
```

You might expect that the parent process has now returned from its wait loop and can thus complete its execution since its children have completed. However, the process command shows that it is still executing.

```
(host:10) > process(host)
 Context            State        Reason     Location       Procedure
 =======            =====        ======     ========       =========
 (host)             Executing
(host:10) >
```

The reason for this is that the child processes have not actually exited yet. The debugger automatically stops a process as it enters the exit code so that it might be examined before it disappears. Thus, the child processes must be continued one more time so that they execute the exit code—after which the parent is notified of their completion. Note that you cannot use the **wait** command in this case, because the exited processes no longer exist and are not a valid context anymore. The **process** command can be used to show that they have exited.

```
(host:10) > continue
(host:10) > process(host:10)
 Context            State        Reason     Location       Procedure
 =======            =====        ======     ========       =========
*(host:10)          Exited       37
ipd > context(0:10)
(0:10) > continue
(0:10) > process(0:10)
 Context            State        Reason     Location       Procedure
 =======            =====        ======     ========       =========
*(0:10)             Exited       37
```

Now the parent process has returned from waiting on the children and has reached the exit routine itself. One more **continue** for this process and it too will exit.

```
ipd > context(host)
(host) > process
 Context           State       Reason    Location       Procedure
 =======           =====       ======    ========       =========
*(host)            Exiting                              __exit()
0x0002c920(host) > continue
(host) > process(host)
 Context           State       Reason    Location       Procedure
 =======           =====       ======    ========       =========
*(host)            Exited      0
ipd >
```

Source Code Examples

PARENT.C

```
#include <nx.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main(argc, argv)
int   argc;
char *argv[];
{
    long node_list[2];
    long pid_list[2];
    long partition_size;
    long return_val;

    /* establish partition in .compute */
    partition_size = nx_initve( NULL, 0, NULL, &argc, argv );
    if ( partition_size == -1 ) {
        perror( “nx_initve” );
        exit( 1 );}

    /* create list of nodes to fork onto */
    node_list[0] = node_self();
    node_list[1] = 0;

    /* fork the processes */
    fflush(stdout);
    return_val = nx_nfork( node_list, 2, 10, pid_list );
    if ( return_val == 0 ) {
        printf( " Child forked...\n" );
        fflush( stdout );
        if ( mynode() == numnodes() ) {
            execl( "./child_host", "./child_host", NULL );}
        else {
            execl( "./child_node", "./child_node", NULL );}}
    else if( return_val == -1 ) {
        perror( "nx_nfork" );
        exit( 1 );}
    else {
        printf( "Hello from Parent (pid %d)\n", getpid() );
        fflush(stdout);
        nx_waitall();}}
```

CHILD_HOST.C

```
#include <stdio.h>

main()
{
   printf ( "Hello from Host Child process (%d:%d)\n", mynode(), myptype() );
}
```

CHILD_NODE.C

```
#include <stdio.h>

main()
{
   printf ( "Hello from Node Child process (%d:%d)\n", mynode(), myptype() );
}
```

Index