ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-204

Nexus User's Guide

by

Ian Foster, John Garnett, and Steven Tuecke*

Mathematics and Computer Science Division

Technical Memorandum No. 204

February 1995
DRAFT

This work was support by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

*Dept. of Computer Science, California Institute of Technology, Pasadena, CA 91125.

Contents

Nexus User's Guide

Ian Foster John Garnett Steven Tuecke

Abstract

This document describes the command line arguments and other features that are common to any program that incorporates the Nexus runtime library. It is divided into sections describing the execution, debugging, tuning, and profiling of such programs. It is not intended as a compiler writer's guide, and does not include any information on the Nexus interface or other internal details.

1 Introduction

Nexus is a runtime library designed primarily as a compiler target for langages supporting task-parallel and mixed data- and task-parallel execution. The Nexus interface and Nexus design are described elsewhere [1, 2]; here, we provide the information needed to execute programs that use Nexus services.

We term a compiler that targets the Nexus runtime system a *Nexus compiler*, and an executable built with such a compiler a *Nexus executable*. A Nexus executable can be passed command line arguments intended for the executable (*executable flags*) and for Nexus itself (*Nexus flags*). Nexus flags follow executable flags and are separated by a separator. The separator used is Nexus compiler dependent. Throughout this document we use `-nexus`. The Nexus flags that are available to a given executable can be determined by typing

```
<app> -nexus -help
```

where `<app>` denotes the executable name. The Nexus flags available may vary according to the operating system, communication protocols, etc., used by Nexus on a particular machine.

Nexus ensures that the executable is not aware of Nexus flags specified on the command line. For example, suppose that the executable accepts a flag `-f` *filename*. Then the command

```
<app> -f filename -nexus -pause_on_fatal
```

runs the executable with the number of arguments set to 3, argument `#0` set to `<app>`, argument `#1` set to `-f`, and argument `#2` set to *filename*. It also passes the flag `-pause_on_fatal` to Nexus (causing Nexus to pause upon a fatal error); however, this flag is not visible to the application.

Table 1 lists the computers and operating systems on which Nexus is currently implemented. Nexus does not yet support computations on networks of heterogeneous machines. This restriction will be removed soon.

Table 1: Nexus Availability

Computer	Operating System	Other Requirements
HP	HPUX 9.x	DCE threads
IBM RS6000	AIX 3.2.x	DCE threads
IBM SP	AIX 3.2.x	DCE threads
Intel Paragon	OSF/1	
Sun	SunOS 4.1.x	
Sun	Solaris 2.x	

Table 2: Workstation-specific Flags

Flag	Description
`-n` N	Create N nodes on default processor
`-nodes` S	Create nodes according to nodelist S
`-Csave_fds` N	Reserve N file descriptors

Tables 2–7 summarize the flags supported by Nexus implementations on different machines. An S means that a string argument is expected. An N means that an integer argument is expected. In the following, we discuss both these flags and the techniques used to execute, debug, profile, and tune Nexus executables.

2 Executing a Nexus Application

Nexus supports several methods for starting distributed parallel programs. Since different Nexus compilers prefer different methods, be sure to check compiler documentation for language-specific details. The startup method can also vary according to the type of parallel computer and network in use.

Table 3: Paragon-specific Flags

Flag	Description
`-sz` N	Number of nodes requested
`-on` N	Single node to load program on
`-noc` N	Specify number of communication partners
`-mbf` N	Specify buffer space for application
`-mea` N	Specify size of dedicated buffers
`-pkt` N	Specify message packet size

Table 4: General-purpose Debugging Flags

Flag	Description
`-debug_command` *S*	Start each context in debugger (if supported)
`-debug_display` *S*	Display for debugger windows (if supported)
`-pause_on_fatal`	Pause on fatal and display id
`-pause_on_startup`	Pause on startup and display id
`-nostart`	Print commands without startup
`-no_catching`	Disable catching of signals
`-catch_sigtrap`	Enable catching of `TRAP` signal
`-catch_fpe`	Enable catching of floating point exception

Table 5: General-purpose Profiling and Tuning Flags

Flag	Description
`-poll_check` *N N*	Report polling frequency data
`-poll_trace`	Generate stack trace on polls (AIX only)
`-profile`	Generate SDDF trace file
`-prsr` *N*	Control profiling level
`-pablofile` *S*	Specify output file for profile data
`-Dnexus` *N*	Enable tracing of Nexus internals
`-hash` *N*	Set handler hash table size
`-rsr_hash` *N*	Set RSR profiling hash table size

Table 6: Pthreads-specific Flags

Flag	Description
`-sched fifo`	FIFO scheduling discipline
`-sched rr`	Round-robin scheduling discipline
`-stack` *N*	Set the stack size of all threads

Table 7: Solaris-specific Flags

Flag	Description
`-concurrency_level` *N*	Control number of active threads

2.1 Naming Nexus Nodes

Before learning how to start a Nexus computation on a particular set of computing resources, it is necessary to know how Nexus names computing resources (processors or computers). A Nexus *node* (virtual processor) is uniquely identified by a pair with the following general form.

node_name`#`*node_number*

The character string *node_name* identifies a particular computing resource or set of resources, while the *node_number* is an integer specifying a member of the *node_name* resource set. The *node_number* can be virtual in the sense that multiple *node_number*s may map to the same physical processor. In this case a cyclic mapping is used to associate virtual *node_number*s with physical processors. The ability to map multiple Nexus nodes onto a single computer allows some flexibility in load balancing decisions. For example, a distributed computation may need to map twice as much work to a two processor workstation in order to use it effectively. One approach is to create a single Nexus node on this computer, to which twice as much work is given. An alternative approach is to treat the workstation as two Nexus nodes; this has the advantage of not requiring changes to work allocation algorithms.

On a uniprocessor workstation, *node_name* is the workstation name and *node_number* can be any integer ≥ 0, with each unique number denoting a unique Nexus node. For example, on a workstation named `dalek`, `dalek#0` and `dalek#1` denote two Nexus nodes located on the same processor.

A symmetric multiprocessor (SMP) incorporates more processors than a uniprocessor workstation. However, on most SMPs the existence of additional processors is transparent to the user, in that we cannot request execution on a particular processor. Hence, when we refer to Nexus nodes `mysmp#0` and `mysmp#1` of a multiprocessor workstation `mysmp`, we are not referring to specific processors but to virtual nodes that will be mapped to available physical processors by the SMP operating system.

2.2 Executing On a Network of Workstations

We first describe how to start a Nexus executable on a network of workstations. Nexus always creates a single *initial node* on the host computer: that is, the computer on which the Nexus command is issued. The flags `-n` and `-nodes` can be used to create additional nodes, as follows.

1. The `-nodes` flag has the general form

   `-nodes` *nodelist*

   where *nodelist* is a colon-separated list of node specifiers, where a node specifier has the general form:

   node_name`[#`*node_number*`][,`*count*`]`

   and where:

node_name : A string, specifying the host name.

`#`*node_number* : (Optional) An integer specifying the number within the node. If omitted, then it defaults to 0.

`,`*count* : (Optional) Number of nodes to start on the given machine, starting with *node_number*. If omitted, then it defaults to 1.

Hence, for example, a node specifier of `dalek#2,2` means to create nodes `dalek#2` and `dalek#3`. A node specifier of `dalek` means to create `dalek#0`. A node specifier of `dalek,3` means to create `dalek#0`, `dalek#1`, and `dalek#2`.

2. The `-n` flag provides a more abbreviated notation that can be used when wish to create nodes only on the host computer. The flag:

   `-n` *number_of_nodes*

   starts *number_of_nodes* nodes, including the initial node, on the host computer.

As an example, execution of the command

```
<app> -nexus -nodes "dalek#1,2:zipper,2:pelican"
```

on a machine called `dalek` creates six nodes: the initial node `dalek#0`, two nodes `dalek#1` and `dalek#2` defined by the first node specifier, two nodes `zipper#0` and `zipper#1` defined by the second node specifier, and a single node `pelican#0` defined by the third node specifier.

It may be necessary to use fully qualified hostnames if the hosts are not all in the same domain.

Note that some Nexus compilers may not use the `-n` or `-nodes` flags. Instead, they may start execution as a single node and then use language constructs to create additional nodes dynamically, during program execution.

Requirements for Node Creation: Nexus can create nodes on other computers, whether dynamically during program execution or at program startup as described by a `-nodes` flag, only if:

1. the `rsh` command works from the initial host to each host in *nodelist*, and
2. each of the hosts shares a common filesystem (via NFS, AFS, etc.) with the initial host (so that the user program may be invoked in the same directory on each machine).

To ensure that `rsh` works as Nexus expects, be sure that one of the files `$HOME/.rhosts` or `/etc/hosts.equiv` contains an entry for each host participating in the computation (one host per line). Note that for security reasons some sites do not allow `.rhosts` files. At those sites ask the system administrator to add the desired hosts to the `/etc/hosts.equiv` file.

2.3 Executing on the Intel Paragon

To both acquire a partition of size N on an Intel Paragon, and to create a single node on every processor of that partition, we use the following Paragon flag in addition to executable flags and Nexus flags.

`-sz` N

If the `-sz` flag is not used, then all available Paragon compute processors will be allocated to our application. This is not usually desirable, so be sure not to forget the `-sz` flag. Paragon flags may be placed anywhere in the argument list.

To acquire a partition of size N but create just a single node on logical node #0 (the first out of the N processors allocated), we use the following Paragon flags.

`-sz` N `-on 0`

The `-on` flag is used by Nexus compilers that allocate nodes dynamically during program execution.

Paragon Tuning. Other Paragon-specific flags may be used to tune the Paragon runtime system. See in particular those described in the section entitled "Specifying Message-Passing Configuration Parameters" in Chapter 2 of the Paragon System User's Guide. These provide additional information on the interaction between communications performance and runtime parameters. We describe some of these here.

By default, the Paragon assumes that each node may need to communicate with all other nodes. This assumption can lead to an inefficient use of system buffer space. The `-noc` *number_of_comms* flag may be used to specify to the Paragon runtime system a maximum number of communication partners for each node. In addition, the `-mbf` *size* flag may be used to control the amount of buffer space allocated to the application. Note that this buffer uses memory that could otherwise be used by the application for other purposes. The default value for the buffer size is 1 MB.

Other flags that may be of interest are `-mea` *size* and `-pkt` *size*. These flags control the size of the dedicated buffers and the message packet size respectively. By default, half of the buffer space (specified via `-mbf`) is given to dedicated buffers (the number of dedicated buffers per node is controlled via the `-noc` flag). The other half is reserved for large messages.

3 Debugging a Nexus Application

Nexus is available in four flavors; see the documentation for your Nexus compiler for information on how these are selected.

`optimized`	:	No debugging or profiling
`safe`	:	Range checking and other checks enabled
`debug`	:	Debugging enabled (implies `-g`)
`profile`	:	Profiling enabled

The `debug` flavor includes extra consistency checks which may be useful in tracking down bugs that show up when using one of the other flavors. It also allows the state of Nexus applications to be examined using symbolic debuggers. It is a good idea to compile applications with the `-g` flag if the application is to be run under a debugger.

The current implementation of Nexus implements a context as a process. This means that debuggers that work on processes can also be used to debug Nexus contexts. However, Nexus applications may comprise multiple processes running on different processors or hosts. Thus, attaching a debugger to the processes that constitute a Nexus computation can be quite involved. Fortunately, Nexus provides support that makes this easier.

3.1 Examining Process State

If a computer has a debugger and X-Windows or other graphical user interface installed, then debuggers can be invoked with the `-debug_command` and `-debug_display` flags. The first of these has the following general form.

`-debug_command` *debugger*

The *debugger* argument names a script or executable that Nexus will execute instead of a context when Nexus receives a request that would normally create a context. Typically, this program will invoke an appropriate debugger, using a separate window for each context. Nexus makes available three environment variables to *debugger*:

- `NEXUS_DEBUG_CONTEXT`: the filename of the executable for the context
- `NEXUS_DEBUG_ARGS`: the command-line arguments that must be passed to the context named by `NEXUS_DEBUG_CONTEXT` so that it can incorporate itself into the Nexus computation
- `NEXUS_DEBUG_DISPLAY`: the argument that was given to the `-debug_display` flag.

The `debug_display` flag has the general form

`-debug_display` *displayname*

and names the display device (e.g., X-windows display) to which debugger output should be directed.

If X-Windows and the dbx debugger are available, then the command

```
<app> -nexus -debug_command debugger -debug_display troop:0.0
```

can be used to execute the example *debugger* script listed in Figure 1. This script creates a unique instance of dbx with output displayed in a separate window on `troop:0.0` for each context created by the Nexus computation. (Make sure *debugger* has execute permissions and is installed in a directory that is in your `PATH`.) Each window will have a title showing the name of the context being debugged. Each context is initially suspended; typing the command `go` in its window initiates execution. This command is aliased to run the context using the various Nexus flags necessary to incorporate the context into the Nexus computation; in order to see these flags, type `alias go`. It is often useful to set breakpoints before issuing the `go` command. If a given system does not have dbx or X-Windows, the script in figure 1 can easily be modified to use a different debugger and to display to a different windowing system.

```
#!/bin/sh
# Nexus debugger script for AIX 3.2. To modify
# for Solaris 2.3, use "-c source $TMP" instead of
# -c $TMP.
#
XTERM=/usr/bin/X11/xterm
RM="/usr/bin/rm -f"
TMP=/tmp/nexus.debug.$$
TITLE="$0:  dbx $NEXUS_DEBUG_CONTEXT"
trap "$RM $TMP; exit 1" 1 2 15
export NEXUS_DEBUG_CONTEXT
export NEXUS_DEBUG_ARGS
export NEXUS_DEBUG_DISPLAY
echo "alias go 'run $NEXUS_DEBUG_ARGS' " > $TMP
$XTERM -title "$TITLE" -display $NEXUS_DEBUG_DISPLAY \
    -e dbx -c $TMP \
    $NEXUS_DEBUG_CONTEXT
$RM $TMP
```

Figure 1: Example of a debugger script

The `-pause_on_fatal` Flag. Other Nexus flags can be useful when debugging on systems that do not support a grahical user interface. For example, the `-pause_on_fatal` flag causes a context to pause and print its process id upon occurrence of a fatal error. At this point it is often possible to attach a debugger to determine why the process failed. Suppose that our Nexus executable is called `myprogram` and that the process used to implement a Nexus context has a process id of 4321. Then the following commands can be used to attach to the process:

```
AIX 3.2.x    : dbx -a 4321
SunOS 4.1.x  : dbx myprogram 4321
Solaris 2.x  : dbx myprogram
               (dbx) debug -p 4321 myprogram
HPUX 9.x     : dbx - 4321
```

After attaching the debugger use the dbx `where` command to obtain a stack trace showing where the error occurred.

The Free Software Foundation's (GNU) debugger `gdb` is also able to attach to a running process. For example if the executable were named `myprogram` and the process id were 4242, the following commands would attach gdb to the process:

```
% gdb myprogram
...
(gdb) attach 4242
```

```
swift:23284:n0:c23284:t0: Fatal error: Signal 6
swift:23284:n0:c23284:t0: Traceback:
swift:23284:n0:c23284:t0: _nx_traceback
swift:23284:n0:c23284:t0: nexus_silent_fatal
swift:23284:n0:c23284:t0: nexus_fatal
swift:23284:n0:c23284:t0: abnormal_death
swift:23284:n0:c23284:t0: cma__sig_deliver
swift:23284:n0:c23284:t0: cma___sig_sync_term
swift:23284:n0:c23284:t0: ?
swift:23284:n0:c23284:t0: abort
swift:23284:n0:c23284:t0: foo__Fv
swift:23284:n0:c23284:t0: cCpP_main__FiPPc
swift:23284:n0:c23284:t0: main
swift:23284:n0:c23284:t0: ?
```

Figure 2: Example of a traceback. This shows the function call stack, with the call that caused the error on the top.

The Intel Paragon debugger (IPD R1.2) is *not* able to debug multithreaded applications. Since all Nexus applications are multithreaded, it is very difficult to debug Nexus applications on the Paragon. For this reason it is advantageous to develop and debug a Nexus application on a network of workstations before trying it on the Paragon.

3.2 Tracebacks

Some implementations of Nexus (for example, the AIX implementation) show stack tracebacks upon fatal errors without requiring the use of any debugger. A traceback will be shown for each context that terminates abnormally. An example of such a traceback is in Figure 2, which shows that Nexus context 23284 crashed as a result of a function called *foo* calling the `abort()` system call.

Nexus errors tend to cascade since the abnormal termination of a context will cause all connected contexts to terminate and so on. The first traceback that is printed usually indicates the true source of the problem. Subsequent tracebacks are usually of the form shown in Figure 3. This particular traceback shows that thread #1 of process 26113 died in the function `tcp_handler_thread`. This is the Nexus function which handles remote service requests. This traceback is typical for contexts that die as a result of losing a connection to another context.

The `pause_on_startup` Flag. Sometimes a Nexus application will crash in such a way that attaching dbx to it does not give any useful information. The `-pause_on_startup` flag can be useful in such cases. This flag causes each Nexus context to pause by spinning in an infinite loop on a variable named `_nx_pausing_for_startup`. This pause gives time to attach a debugger. Once the debugger is attached, allow the process to continue by using the dbx command `assign _nx_pausing_for_startup=0` and then typing `cont`.

```
swift:26113:n1:c26113:t1: Traceback:
swift:26113:n1:c26113:t1: _nx_traceback
swift:26113:n1:c26113:t1: nexus_silent_fatal
swift:26113:n1:c26113:t1: nexus_fatal
swift:26113:n1:c26113:t1: check_proto_for_close
swift:26113:n1:c26113:t1: select_and_read
swift:26113:n1:c26113:t1: tcp_handler_thread
swift:26113:n1:c26113:t1: thread_starter
swift:26113:n1:c26113:t1: cma__thread_base
```

Figure 3: Another example of a traceback.

The `-nostart` Flag. As an alternative to `-pause_for_startup`, the `-nostart` flag may be used. When this flag is used, no processes are actually started; instead, a message is printed out for each context that shows what command would have been used to start the process if the `-nostart` flag had not been used. This information can be used to determine what command to run from inside dbx to start a given Nexus context.

3.3 Signal handling

By default Nexus traps various signals including `BUS`, `SEGV`, `PIPE`, `ABRT`, and `ILL`. These signals usually cause the `abnormal_death` function to be called. To disable these signals from being trapped, use the Nexus flag `-no_catching`. This ability can be useful when running from inside a debugger since some debuggers have difficultly giving a stacktrace that extends past the signal handler.

Two other signal-related Nexus flags that may prove useful are `-catch_sigtrap` and `-catch_fpe`. The first requests that the `TRAP` signal cause program termination. The second causes the program to terminate upon receipt of a floating point exception signal.

3.4 Tracing Nexus Internals

The `-Dnexus` *level* flag can be useful for those familiar (or wishing to become familiar) with Nexus internals. The integer *level* specifies the level of detail that Nexus debug statements should print. The lowest level of detail is given by `-Dnexus 1` and the highest is limited by the size of a machine integer. The `-Dnexus` flag has an effect only if the `debug` flavor of Nexus is used.

3.5 Using Testcenter and Purify

Commercial debugging tools with graphical interfaces, such as Testcenter and Purify, can be used to debug networked Nexus applications if the user's `$HOME/.cshrc` file contains a line setting the DISPLAY environment variable to point to the proper host.

4 Tuning a Nexus Application

Several Nexus flags may be used to tune the performance of a Nexus application. Some of these flags are specific to a particular implementation of Nexus.

Reserving File Descriptors. Network versions of Nexus support the `-Csave_fds` *integer* flag. This may be used to instruct Nexus to reserve the specified number of file descriptors for use by the user's application. By default ten file descriptors are saved. The system saves a minimum of three file descriptors even if `-Csave_fds` is used to specify fewer. This flag may be necessary in a large application (i.e. many nodes and/or contexts) that must have many files open simultaneously.

Solaris Concurrency Level. The Solaris version of Nexus supports a flag with the following general form.

`-concurrency_level` *integer*

The *integer* value is used as a parameter to the Solaris `thr_setconcurrency` system call. By default the Solaris threads system ensures that a sufficient number of threads are active so that the process can continue to make progress. Nexus can use the `thr_setconcurrency` call to instruct the system to ensure that a larger number of threads can be active at one time (at the expense of system resources). This can result in increased performance. See the Solaris `thr_setconcurrency()` man page for more information. By default, Nexus uses a Solaris concurrency level of 5.

Setting Thread Stack Size. Versions of Nexus that use pthreads (POSIX threads) support the `-stack` *stack_size* argument, which sets the stack size of all threads to the integer *stack_size*. (These versions include AIX and HP-UX; the Paragon uses an older version of pthreads that does not allow stack size to be changed.) Applications that use many automatic variable may require this flag.

Pthreads Scheduling Discipline. Versions of Nexus that use pthreads also support the `-sched fifo` and `-sched rr` flags. The `-sched fifo` flag causes Nexus to use a FIFO (first in, first out) scheduling discipline for threads. The `-sched rr` flag causes Nexus to use a round robin scheduling discipline for threads. Round robin is the default. The round robin discipline uses timeslicing to share the processor among threads. The FIFO discipline allows a thread to run until it completes or blocks before scheduling another thread. Round robin usually imposes more overhead (because of timer interrupts for implementing the timeslicing) but for some applications it gives better performance.

Polling Diagnostics. For certain applications, it may be useful to know how often a given Nexus context checks for the arrival of messages from other contexts. The Nexus flags `-poll_check` *min max* and `-poll_trace` can be used to gain some insight on this matter. The `-poll_check` flag causes a message to be printed out if the time between `nexus_poll()` calls does not fall in the supplied range, *min–max*. Both *min* and *max* should be real numbers (in units of seconds). In the AIX version of Nexus, the `-poll_trace`

flag causes a stack traceback to be printed each time `-poll_check` causes a notice to be printed; this allows the programmer to identify the points in the program where polls are infrequent.

5 Profiling a Nexus Application

Nexus can produce trace data for the Pablo performance visualization tool developed by the Picasso group at University of Illinois. Pablo source and documentation are available via anonymous ftp from `bugle.cs.uiuc.edu` in the directory `/pub/Release`. Building Pablo requires that both Motif and X11R5 be installed. Pablo is not available as a binary distribution because of Motif licensing restrictions.

The Nexus flags used to produce Pablo tracefiles are `-profile`, `-prsr`, and `-pablofile`. The `-profile` flag enables profiling, the `-pablofile` flag specifies a filename prefix for output files, and the `-prsr` flag determines the remote service request (RSR) data that is recorded:

0	:	log each RSR send and receive
-1	:	keep cumulative profile, and dump only at end (default)
integer > 0	:	keep cumulative profile, and dump after every *integer* RSR receives

If `-prsr` is not specified, the profile level defaults to 1. A typical usage of these flags is as follows:

```
<app> -nexus -profile -prsr 2 -pablofile app.1
```

Trace data is saved for each Nexus context that is part of the computation. The Pablo data is dumped in Self Describing Data Format (SDDF) in one file per context. Each file is prefixed with the string specified via the `-pablofile` flag, and has a unique suffix based on node and context identifiers.

Tracefiles must be merged into a single file before they can be examined with Pablo. This merging can be accomplished by using a script named `allmergeSDDF` that is distributed with Nexus, as follows

`allmergeSDDF` *filename*

where *filename* is the filename prefix supplied to the Nexus executable with the `-pablofile` flag.

The `allmergeSDDF` command creates several files including *filename*`_all.status`, *filename*`_all.binary`, and *filename*`_all.ascii`. To invoke Pablo, type:

```
runPablo
```

after which the file *filename*`_all.binary` can be loaded into the visualization tool. Note that Pablo requires that a configuration file be created for each tracefile that is to be visualized. Pablo can use the same configuration file for different tracefiles but only if the tracefiles were generated by the same (or very similar) executable. Study the examples that come with Pablo in order to learn how to create Pablo configuration files tailored to the the tracefiles produced by a given executable. Pablo problems may be reported to `pablo-bugs@guitar.cs.uiuc.edu`.

6 Obtaining Nexus and Nexus Compilers

The Nexus source code, additional documentation, and help can be obtained from the following ftp site, World Wide Web URL, and email address:

ftp : `ftp.mcs.anl.gov`, directory `/pub/nexus`
http : `http://www.mcs.anl.gov/nexus`
email : `nexus@mcs.anl.gov`

Fortran M is a compiler that uses Nexus as a runtime library. Information about Fortran M is available at the following locations:

ftp : `ftp.mcs.anl.gov`, directory `/pub/fortran-m`
http : `http://www.mcs.anl.gov/fortran-m`
email : `fortran-m@mcs.anl.gov`

Compositional C++(CC++) is a compiler that uses Nexus as a runtime library. Information about CC++is available at the following locations:

ftp : `compbio.caltech.edu`, directory `/pub/C++`
http : `http://compbio.caltech.edu`
email : `cc++-requests@compbio.caltech.edu`

References

[1] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, Nexus: An interoperability toolkit for parallel and distributed computer systems, Technical Report ANL/MCS-TM-189, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[2] I. Foster, C. Kesselman, and S. Tuecke, Nexus: Runtime Support for Task-Parallel Programming Languages, Mathematics and Computer Science Division, Argonne National Laboratory, ANL/MCS-TM-205, February 1995 (draft).