M++ User’s Manual
(version 1.00)

Munehiro Fukuda
Computing and Software Systems
University of Washington, Bothell
18115 Campus Way, N.E.
Bothell, WA 98011
email: mfukuda@u.washington.edu

Naoya Suzuki
Information Sciences and Electronics
University of Tsukuba,
1-1-1 Ten’nohdai
Tsukuba, Ibaraki 305, JAPAN
email: nas@is.tsukuba.ac.jp

September 1, 2002
# Contents

1 Preface 5

2 Porting and Accessing the System 6
   2.1 Availability ................................................. 6
   2.2 Installing M++ .............................................. 6
      2.2.1 Downloading ........................................... 7
      2.2.2 Compiling gcc-2.95 .................................. 7
      2.2.3 Setting your shell environment ...................... 8
      2.2.4 Compiling M++ ........................................ 9
      2.2.5 Using GM library .................................... 10
   2.3 Porting M++ .................................................. 11
   2.4 Additional Notes .......................................... 11

3 Getting Started 12
   3.1 M++ Profile ................................................ 12
   3.2 M++ Thread Compilation .................................. 13
   3.3 System Invocation/Termination ........................... 15
   3.4 Using Myrinet Zero-Copy Communication ................ 18

4 Writing M++ Threads 19
   4.1 Execution Model ........................................... 19
   4.2 Network Components ...................................... 22
      4.2.1 The daemon Class .................................... 22
      4.2.2 The daemonobj Class ................................. 23
      4.2.3 Nodes .................................................. 24
      4.2.4 Links .................................................. 27
      4.2.5 Classes Used for Network Components .............. 31
   4.3 Threads ...................................................... 31
      4.3.1 Thread Definition .................................... 32
1 Preface

M++ is a new cluster computing paradigm for multi-agent applications viewed as the interactions among autonomous objects, each behaving with its own goals and reacting to the local information in a synthetic world. M++ self-migrating threads are C++ compiled objects, capable of constructing logical computational networks and autonomously migrating themselves over a cluster system. They represent a natural mechanism of implementing multi-agent applications: each individual agent can be represented as a distinct member of collection of self-migrating threads, moving through a logical network representing a synthetic world and interacting with one another. Several actual applications M++ assumes are:

- **Ecological simulation**: simulate a transient population of preys and predators such as fish and sharks [Dew84], and find the genetic code of the most efficient ant forager through a genetic algorithm having many ants searching for food in a farm [CJ92].

- **Artificial life**: simulate the evolution and learning process of neural network for solving a given problem [NNKdG98], and observe the growth of artificial plants [PL96].

- **Complex business modeling**: decide the best cargo-routing strategy by finding which local interactions lead to global behaviors in cargo operations [Wak01].

For more details on M++ suitability to those multi-agent applications, please refer to the following three research papers: [SFB99], [FSCK01], and [LMF+02]. Also, you can obtain further acquaintance with multi-agent applications through the readings: [Fer99] and [Wei99].

This manual provides the procedures necessary for users to start programming M++ self-migrating threads, (simply called M++ threads in the following explanation.) To run M++ threads, you need to install the system, start an M++ daemon at each PC or workstation, code your M++ threads in the M++ language, compile them, and inject the executables to one of active daemons. The manual introduces system installation, daemon invocation, the M++ language specification, thread compilation, and trouble shooting. It also includes code examples to prompt users' understanding.

We welcome any comments and suggestions regarding M++, please contact us at:

Munehiro Fukuda  mfukuda@u.washington.edu for English assistance
Naoya Suzuki  nas@is.tsukuba.ac.jp for Japanese assistance
2 Porting and Accessing the System

2.1 Availability

The currently available platforms to run M++ include:

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Operating Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Intel Solaris 7 and 8, Redhat Linux 7.3</td>
</tr>
<tr>
<td>Sparc</td>
<td>SunOS 5.5.1-SunOS 5.8</td>
</tr>
<tr>
<td>AP3000</td>
<td>SunOS 5.6</td>
</tr>
</tbody>
</table>

Table 1: Available Platforms

We have installed M++ at the following sites. Note that *** in Table 2 means a gateway’s or each cluster node’s IP name. This manual does not disclose this information for security reasons. To find the appropriate name, ask your local system owner or manager. If you are a user at any of those sites, you do not have to install the system. Please skip to Section 2.2.3 to set up your shell environment.

<table>
<thead>
<tr>
<th>Sites</th>
<th>Platform</th>
<th>OS</th>
<th>IPs and System Managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW Bothell</td>
<td>PCs</td>
<td>Redhat Linux</td>
<td>medusa.bo.thell.washington.edu</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:dgrimmer@uwb.edu">dgrimmer@uwb.edu</a></td>
</tr>
<tr>
<td>Univ. of Tsukuba</td>
<td>PCs</td>
<td>Solaris 8</td>
<td>howdy00.padc.mmpc.is.tsukuba.ac.jp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:nas@is.tsukuba.ac.jp">nas@is.tsukuba.ac.jp</a></td>
</tr>
<tr>
<td>UC Irvine</td>
<td>Sparc</td>
<td>SunOS 5.5.1-5.8</td>
<td>***.ics.uci.edu</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:support@ics.uci.edu">support@ics.uci.edu</a></td>
</tr>
<tr>
<td>Ehime University</td>
<td>AP3000</td>
<td>SunOS 5.6</td>
<td>***.cs.ehime-u.ac.jp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:kob@cs.ehime-u.ac.jp">kob@cs.ehime-u.ac.jp</a></td>
</tr>
</tbody>
</table>

Table 2: Available Sites

To install M++ on the platforms listed in Table 1, please follow the instructions given in Section 2.2. If you are interested in porting to the other platforms, please refer to Section 2.3.

2.2 Installing M++

First of all, please make sure that your cluster is NFS-mounted. The same disk image must be visible to all computing nodes of the cluster. In addition, each computing node must have an independent IP address, (whether it is a private or a public IP). Unless these requirements are satisfied, the installation is impossible.
2.2.1 Downloading

To obtain a gzipped tar file of M++, visit any of the following M++ web pages:

<table>
<thead>
<tr>
<th>Sites</th>
<th>Web Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>UW Bothell</td>
<td><a href="http://faculty.washington.edu/mfukuda/m++">http://faculty.washington.edu/mfukuda/m++</a></td>
</tr>
<tr>
<td>Univ. of Tsukuba</td>
<td><a href="http://www.padc.mmpc.is.tsukuba.ac.jp/member/naoya/m++">http://www.padc.mmpc.is.tsukuba.ac.jp/member/naoya/m++</a></td>
</tr>
<tr>
<td>UC Irvine</td>
<td><a href="http://www.ics.uci.edu/">http://www.ics.uci.edu/</a> mfukuda/m++</td>
</tr>
</tbody>
</table>

Table 3: M++ Web Pages

Download, unzip, and extract the file.

```
% gzip -d mpp.tar.gz | tar -xf -
```

Once you have extracted the M++ archive or if you are a user at a site where M++ has been installed, you will find the M++ directory that is organized as follows:

<table>
<thead>
<tr>
<th>Directories</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>m++/bin/</td>
<td>M++ executables (in the near future)</td>
</tr>
<tr>
<td>m++/src/</td>
<td>M++ daemon source and executables</td>
</tr>
<tr>
<td>m++/src/translator/</td>
<td>M++ compiler source and executables</td>
</tr>
<tr>
<td>m++/src/GMW1.xx/</td>
<td>GM zero-copy library (GMW1.11 for Solaris and 1.20 for Linux)</td>
</tr>
<tr>
<td>m++/sample/</td>
<td>Sample code</td>
</tr>
<tr>
<td>m++/appl/</td>
<td>Applications</td>
</tr>
<tr>
<td>m++/doc/</td>
<td>All manuals and papers</td>
</tr>
</tbody>
</table>

Table 4: Directory information

All executable codes will be generated in m++/src and m++/src/translator after installing M++.

2.2.2 Compiling gcc-2.95

The current implementation of M++ system is based on gcc-2.95.1 through to 2.95.1. If your site uses a newer version of gcc such as gcc-3.x, download gcc-2.95.3 and compile it.

1. Version Check

Type `gcc -v` to see which version of gcc you are using. If it is newer than gcc-2.95.3, follow the instructions below, otherwise skip to Section 2.2.3.
2. Downloading
Visit http://gcc.gnu.org/gcc-2.95/, download gcc-2.95.3 to the m++ directory, and extract the files.

3. Compilation
Assuming that your working directory include m++ directory, compile the source code as instructed below:

```
  cd m++/gcc-2.95.3
  configure --prefix=/home/m++/gcc-2.95.3
  make bootstrap
  make install
  cd ..
```

2.2.3 Setting your shell environment

You need to set up your shell environment. Add the following two statements in your .cshrc and reinitialize it by invoking “source ~/.cshrc”.

<table>
<thead>
<tr>
<th>Sites</th>
<th>A Statement to Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>medusa.bothell.washington.edu</td>
<td>set path=/home/m++/gcc-2.95.3/bin $path</td>
</tr>
<tr>
<td></td>
<td>/home/m++ /src /home/m++ /src/translator</td>
</tr>
<tr>
<td></td>
<td>setenv MPPDIR /home/m++ /src</td>
</tr>
<tr>
<td></td>
<td>setenv MPPAR CH LINUX</td>
</tr>
<tr>
<td>howdy00.padc.mmpc.is.tsukuba.ac.jp</td>
<td>set path=/home/m++/gcc-2.95.3/bin $path</td>
</tr>
<tr>
<td></td>
<td>/home/mplus2 /src /home/mplus2 /src/translator</td>
</tr>
<tr>
<td></td>
<td>setenv MPPDIR /home/mplus2 /src</td>
</tr>
<tr>
<td></td>
<td>setenv MPPAR CH SOLARIS_7</td>
</tr>
<tr>
<td>**.ics.uci.edu</td>
<td>set path=/home/m++/gcc-2.95.3/bin $path</td>
</tr>
<tr>
<td></td>
<td>/extra/ mfukuda0 /m++ /src</td>
</tr>
<tr>
<td></td>
<td>/extra/ mfukuda0 /m++ /src/translator</td>
</tr>
<tr>
<td></td>
<td>setenv MPPDIR /extra/ mfukuda0 /m++ /src</td>
</tr>
<tr>
<td></td>
<td>setenv MPPAR CH SOLARIS_2.5</td>
</tr>
<tr>
<td>**.cs.ehime-u.ac.jp</td>
<td>set path=/home/ mes /gcc-2.95.3/bin $path</td>
</tr>
<tr>
<td></td>
<td>/home/ mes /src /home/ mes / src/translator</td>
</tr>
<tr>
<td></td>
<td>setenv MPPDIR /home/ mes /src</td>
</tr>
<tr>
<td></td>
<td>setenv MPPAR CH SOLARIS_2.5</td>
</tr>
<tr>
<td>Other Linux systems</td>
<td>set path= /... /m++ /gcc-2.95.3/bin $path</td>
</tr>
<tr>
<td></td>
<td>... /m++ /src ... /m++ /src /translator</td>
</tr>
<tr>
<td></td>
<td>setenv MPPDIR ... /m++ /src</td>
</tr>
<tr>
<td></td>
<td>setenv MPPAR CH LINUX</td>
</tr>
</tbody>
</table>

Table 5: Path set-up
For your information, a shell variable `$MPPARCH` specifies the underlying OS type as shown below:

<table>
<thead>
<tr>
<th>Values</th>
<th>Machine Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOLARIS_7</td>
<td>PCs under Solaris7 or Solaris8</td>
</tr>
<tr>
<td>SOLARIS_2.5</td>
<td>Sparc and AP3000</td>
</tr>
<tr>
<td>LINUX</td>
<td>PCs under Linux</td>
</tr>
</tbody>
</table>

Table 6: `$MPPARCH`

If you are a user at any of the sites: UWB, Tsukuba, UCI and Ehime, skip to Section 2.4, otherwise go on to the next section to complete your M++ installation.

2.2.4 Compiling M++

Before compiling M++, you need to modify `Makefile` located in the `m++/src` directory. This file include the following commented-out statements:

```c
#SOLARIS_2.5
#LIBS = -lnsl -lsocket -lposix4 -ldl -lpthread -lfl
#SOLARIS_7
#LIBS = -lnsl -lsocket -lposix4 -ldl -lrt -lpthread -lfl
#LINUX
#LIBS = -lnsl -ldl -lrt -lpthread -lfl
```

Each statement corresponds to a different operating system type. Choose the appropriate one and remove the `#` character from the line head, so that this line will get effective. If your system is PC Linux, you should choose:

```c
#LINUX
LIBS = -lnsl -ldl -lrt -lpthread -lfl
```

Now, assuming that your working directory includes the `m++` directory in it, compile the source code as instructed below:

```bash
% cd m++/src
% make
% cd translator
% make all
% cd ../../...
```
2.2.5 Using GM library

If your cluster has installed a Myrinet card to each computing node, you can compile M++ with the underlying GM library for better performance. If you wish to use the GM library, follow the instruction shown below, otherwise skip to Section 2.4.

1. Obtaining the GM library
   Visit the Myricom homepage, http://www.myri.com and download the appropriate library. Note that the library downloading asks you to type a password received from Myrinet upon your purchase of Myrinet.

2. Installing the library
   Type the following commands to extract the GM library files in the GM home directory on your system:

   ```
gunzip -c gm-1.5.1_Linux.tar.gz | tar xvf -
cd {GM_HOME}
```

   Note that gzipped file name depends on the OS version The above file is valid for Linux. Thereafter, follow the README-linux file for the installation.

3. Modifying m++/src/Makefile
   GM_LIBS = -L{your GM library's directory path}/libgm/
   GM_INCLUDES = -I{your GM library's directory path}/include/
   GMW_LIB = -L{your M++ directory path}/lib/GMW1.20/lib/
   GMW_INCLUDE = -I{your M++ directory path}/lib/GMW1.20/include/
   For instance, medusa.bothell.washington.edu has set up this Makefile as follows:

   ```
   GM_LIBS = -L/usr/src/gm-1.5.1.1_Linux/libgm/
   GM_INCLUDES = -I/usr/src/gm-1.5.1.1_Linux/include/
   GMW_LIB = -L/home/m++/src/GMW1.20/lib/
   GMW_INCLUDE = -I/home/m++/src/GMW1.20/lib/include/
   ```

4. Modifying m++/src/GMW1.20/Makefile
   GM_LIBS = -L{your GM library's directory path}/libgm
   GM_INCLUDES = -I{your GM library's directory path}/include
   For instance, medusa.bothell.washington.edu has set up this Makefile as follows:

   ```
   GM_LIBS = -L/usr/src/gm-1.5.1.1_Linux/libgm
   GM_INCLUDES = -I/usr/src/gm-1.5.1.1_Linux/include
   ```

5. Compiling the M++ source code

   Assuming that your working directory includes the m++ directory in it, type the follow sequence of commands for compilation. After the compilation, the m++/src directory must include thrd_gmw and inject_gmw.
% cd m+/src/GMW1.20
% make
% cd ..
% make gmw
% cd ../..

2.3 Porting M++

Before porting M++, follow the instructions for system installation described in Section 2.2 to download and extract the tar file. Here are several tips to port M++ to your machine.

Hardware-dependent code
- `setjmp()` and `longjmp()` to capture and resume register values
- the M++ original thread library named `stthread`

Unix-dependent code
- dynamic-linking library, Pthread, and socket communication.

Among them, what you have to pay your attention for includes `setjmp()`/`longjmp()` functions, `stthread` and `pthread`.

The `setjmp()`/`longjmp()` pairs can be found in the files: SelfMigThr.h, thrFunc.h, and Stthread.cpp. Register contents are saved in a `jmpbuf` data structure by `setjmp` and resumed by `longjmp`. You need to know which register content will be saved in which `jmpbuf` element. Also, you must check how a stack grows. With these knowledge, modify those three cpp files.

The latter can be located in many files, however of importance is port.cpp. This code handles multiple sockets concurrently using `pthread`. Check the scheduling policy of your `pthread` library. If the library does not always force each thread to relinquish CPU whenever it is blocked on an I/O operations, the system will hang up. In that case, modify port.cpp so that no `pthread` will keep holding CPU upon an blocked I/O.

2.4 Additional Notes

1. If you want to run sample programs and/or applications, you are also required to visit each directory and to recompile the source code. Follow Section 3.2.

2. If you are a user at any of specific sites: UW Bothell, Univ. Tsukuba, UC Irvine and Ehime Univ., the following platform information may be useful.
(a) **Specification of UW Bothell’s Medusa Cluster**

<table>
<thead>
<tr>
<th>Spec. items</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>PC (dual-i686 1.4GHz with 256KB cache/512MB main memory)</td>
</tr>
<tr>
<td>$\eta$nodes</td>
<td>6 (medusa, mnode1 - mnode5)</td>
</tr>
<tr>
<td>Interconnection</td>
<td>4Gbps Myrinet [BCF+95]</td>
</tr>
<tr>
<td>File</td>
<td>NFS mounted to medusa’s 34GB file system</td>
</tr>
<tr>
<td>External connection</td>
<td>medusa only</td>
</tr>
</tbody>
</table>

(b) **Specification of Tsukuba Univ.’s Howdy Cluster**

<table>
<thead>
<tr>
<th>Spec. items</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>PC (Athlon 1Ghz with DDR-SDRAM 256M-byte memory)</td>
</tr>
<tr>
<td>$\eta$nodes</td>
<td>8 (howdy00 - howdy07)</td>
</tr>
<tr>
<td>Interconnection</td>
<td>1.28Gbps Myrinet [BCF+95]</td>
</tr>
<tr>
<td>File</td>
<td>NFS mounted to howdy00’s 6.4GB file system</td>
</tr>
<tr>
<td>External connection</td>
<td>howdy00 only</td>
</tr>
</tbody>
</table>

(c) **Specification of UW Irvine’s Iris Cluster**

<table>
<thead>
<tr>
<th>Spec. items</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Sun Sparc Station SS5 with 96M-byte memory</td>
</tr>
<tr>
<td>$\eta$nodes</td>
<td>9 (iris00 - iris09)</td>
</tr>
<tr>
<td>Interconnection</td>
<td>100Mbps Ethernet</td>
</tr>
<tr>
<td>File</td>
<td>NFS mounted to hippocrates’ 1.6GB file system</td>
</tr>
<tr>
<td>External connection</td>
<td>Any nodes available from other ICS workstations</td>
</tr>
</tbody>
</table>

(d) **Specification of Ehime Univ.’s AP3000 Cluster**

<table>
<thead>
<tr>
<th>Spec. items</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>UltraSPARC 167MHz with X-byte memory</td>
</tr>
<tr>
<td>$\eta$nodes</td>
<td>25 (apnet0000 - apnet0024)</td>
</tr>
<tr>
<td>Interconnection</td>
<td>1.6Gbps AP-Net</td>
</tr>
<tr>
<td>File</td>
<td>NFS mounted</td>
</tr>
<tr>
<td>External connection</td>
<td>Nodes 0, 1, 2, 3, 4, 8, 9, 10, 11, 16, 17, 18, and 19</td>
</tr>
</tbody>
</table>

3 Getting Started

3.1 M++ Profile

Upon initialization, an M++ daemon reads a configuration file. This file must be located as *profile* in each user’s current working directory, (where the daemon is invoked.) It declares the following items:
<table>
<thead>
<tr>
<th>Items</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>A socket port number used for thread migration.</td>
</tr>
<tr>
<td></td>
<td>No default port. It must be given by a user.</td>
</tr>
<tr>
<td>hostnum</td>
<td>The number of PCs constituting the system.</td>
</tr>
<tr>
<td></td>
<td>It must be 1 or larger.</td>
</tr>
<tr>
<td>mode</td>
<td>A thread library name: pthread or sthread.</td>
</tr>
<tr>
<td></td>
<td>Pthreads are default. Streads are our original threads.</td>
</tr>
<tr>
<td>ip name and daemon ID</td>
<td>Each PC’s IP name and its corresponding daemon ID.</td>
</tr>
<tr>
<td></td>
<td>The daemon ID must be a unique non-negative integer.</td>
</tr>
</tbody>
</table>

The following profile defines a configuration in that an M++ daemon runs on howdy00, 01, 02, and 03, communicates with the other daemons through port # 92767, and simulates your agents with Pthreads.

```plaintext
port 92767
hostnum 4
mode pthread
describe daemonid
howdy00 0
howdy01 1
howdy02 2
howdy03 3
describe_end
```

### 3.2 M++ Thread Compilation

Prior to an injection, each M++ self-migrating thread must be compiled into executable code with the M++ compiler. The M++ compiler actually translates your M++ code into C++ programs and thereafter passes them to the g++ compiler. It accepts the following eight translation options:

<table>
<thead>
<tr>
<th>options</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>-translate</td>
<td>translate M++ code to the corresponding C++ code and save it in files</td>
</tr>
<tr>
<td>-I</td>
<td>specify an include directory</td>
</tr>
<tr>
<td>-L</td>
<td>specify a library directory</td>
</tr>
<tr>
<td>-l</td>
<td>specify a library file</td>
</tr>
<tr>
<td>-D</td>
<td>make a given macro effective</td>
</tr>
<tr>
<td>-U</td>
<td>make a given macro ineffective</td>
</tr>
<tr>
<td>-E</td>
<td>preprocess M++ code and print out the result</td>
</tr>
<tr>
<td>-v</td>
<td>print out the version/translation information while compiling M++ code into the executable</td>
</tr>
</tbody>
</table>

For example, assume that your thread is coded in the mythread.mpp source file in the current working directory. If it refers to the math library, you can compile
mythread.mpp to the mythread executable code by typing:

```
% m++ mythread.mpp -lm
```

Currently, the M++ compiler has two restrictions:

1. A .mpp source code takes a different keyword to incorporate its header files. It is the `import{ }` keyword.
   - `import "Filename" { class/struct/union prototype declarations }` incorporates class/struct/union data structures from a user-defined Filename. If you incorporate only functions and macros, leave blank between this pair of blankets `{ }`. However, note that `{ }` must not be omitted.
   - `import <Filename> { class/struct/union prototype declarations }` incorporates class/struct/union data structures from a system-provided Filename. The notation, `import <Filename> { }` incorporates only the functions and macros of Filename.

Example:
```
import <cmath> {}
import <cstdlib> {}
import "Timer.h" { class Timer; }
```

2. The following keywords are reserved:

- along
- asm
- auto
- bool
- break
- case
- catch
- char
- class
- clocking
- const
- const_cast
- continue
- create
- daemon
- daemonobj
- default
- delete
- destroy
- double
- dynamic_cast
- edestroy
- explicit
- extern
- efork
- eforkalong
- float
- for
- fork
- forklalong
- friend
- goto
- gvt_delta
- gvt_end
- gvt_ready
- gvt_start
- gvt_time
- hop
- hopalong
- if
- import
- inline
- int
- link
- long
- mpp
- mutable
- namespace
- new
- node
- operator
- private
- protected
- public
- register
- reinterpret_cast
- return
- short
- signed
- sizeof
- sleep
- static
- static_cast
- struct
- switch
- template
- terminate
- this
- thread
- thread_id
- throw
- to
- try
- true
- typedef
- typeid
- typename
- union
- unsigned
- using
- virtual
- void
- volatile
- wakeup
- wchar_t
- while
- with

For example, consider compilation of `m++/sample/Car/Car.mpp`. This file includes three types of M++ threads:

14
<table>
<thead>
<tr>
<th>Thread names</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCashier</td>
<td>work as a gas station cashier who pumps gas into a tank</td>
</tr>
<tr>
<td>SportsCar</td>
<td>function as a sports car that gets regular fuel from a GCashier</td>
</tr>
<tr>
<td>Truck</td>
<td>function as a truck that receives diesel fuel from a GCashier</td>
</tr>
</tbody>
</table>

Compiling `m++/sample/Car/Car.mpp` is very simple. Copy the file to your current working directory and just type “`m++ Car.mpp`”. The compiler will produce three independent executables: `GCashier`, `SportsCar`, and `Truck`, which are then ready to be injected to one of active M++ daemons.

### 3.3 System Invocation/Termination

#### Invocation

You must manually invoke an M++ daemon from all computing nodes that are to run your application. For each machine, follow the invocation procedure shown below:

1. Open an `xterm` or `tera-term` window.
2. Login the gateway or the master node of your cluster with ssh,  
   (e.g., `medusa.bothell.washington.edu at UW Bothell`).
3. Login each machine with rsh.
4. Type `thr (or thrd & for a background execution)`.

The M++ daemon works as a simple Unix user process. It runs without being idle whether or not it has received active M++ threads to execute. If you debug your M++ threads or must share computing nodes with other users, you may want to start a daemon with a lower priority through a Unix `nice` option:

```
% nice 20 thrd&
```

When all the participating computing nodes have started a daemon, these daemons synchronize with one another and thereafter display the following message. (The example shows the message when four daemons are invoke.)

```
% thrd
port : 92767
Connection established to howdy00 with daemon 0
Connection: Wait for howdy01 acceptance ...
```
Connection established to howdy01 with daemon 1
Connection: Wait for howdy02 acceptance ...
Connection established to howdy02 with daemon 2
Connection: Wait for howdy03 acceptance ...
Connection established to howdy03 with daemon 3
Total daemons : 4
Mode : pthread
Stack : 65536 bytes
Ready for being injected.

At this moment, you are ready to inject M++ threads.

**Injection**

The `inject` command injects new M++ threads to the system. The input format of this command is:

```
% inject hostname filename #threads arg1 arg2 arg3 ....
```

The `hostname` parameter is the IP name of a target daemon where you want to inject M++ threads. The `filename` parameter specifies the thread executable code. The `#threads` parameter gives the number of threads to instantiate from `filename` and to inject into that daemon. Arguments such as `arg1`, `arg2`, and `arg3` are passed to each of those threads in form of a string array. This argument-passing mechanism is detailed in Section 4.

The following example injects M++ threads into `howdy00` to run the Car sample program.

```
mpplus2@howdy00[1]$ thrd&
[1] 18170
port : 10145
Connection established to howdy00 with daemon 0
Total daemons : 1
Mode : sthread
Stack : 32768 byte
Ready for being injected.

mpplus2@howdy00[2]$ inject howdy00 ./GSCashier 1
Injected 1 ./GSCashier thread(s) to howdy00
GSCashier: Waiting for cars.

mpplus2@howdy00[3]$ inject howdy00 ./Truck 1
```

16
Injected 1 ./Truck thread(s) to howdy00
Truck: I'm at the GasStation.
GSCashier: Injected fuel to Truck
Diesel: 900  Regular: 1000
GSCashier: Waiting for cars.
Truck: I'm filled with fuel.

mplus2@howdy00[4]% inject howdy00 ./SportsCar 2
Injected 2 ./SportsCar thread(s) to howdy00
SportsCar: I'm at the GasStation.
GSCashier: Injected fuel to SportsCar
Diesel: 900  Regular: 950
GSCashier: Waiting for cars.
SportsCar: I'm filled with fuel.

SportsCar: I'm at the GasStation.
GSCashier: Injected fuel to SportsCar
Diesel: 900  Regular: 900
GSCashier: Waiting for cars.
SportsCar: I'm filled with fuel.

mplus2@howdy00[5]% fg
~naoya/m++/thrd
^Cthrd: Interrupted.
thrd: Closed all connections.
thrd: Terminated.

mplus2@howdy00[6]%

This example assumes that the current directory has the three executable files of the Car sample program: GSCashier, SportsCar, and Truck. A user first invokes a single M++ daemon on howdy00. Second, he/she injects a GSCashier thread that behaves as a gas station cashier, initially prepares 1000 litters of diesel and regular gasoline respectively, and pumps gasoline into a car tank. Third, the user injects a Truck thread that receives 100 litters of diesel fuel from the GSCashier thread. Fourth, he/she injects two SportsCar threads, each receiving 50 litters of regular gasoline.

The inject command displays the following messages when detecting errors in thread injection:

- SharedLib: Cannot load the dynamic module: means that a given file name does not include executable thread code.
- **Segmentation fault**: means that the number of arguments actually given is incompatible to that of arguments coded in the thread program.

They are not fatal errors to M++ daemons. All you need is simply retype a correct file name and arguments.

**Termination**

One daemon termination is enough to kill all the other daemon processes. To do that, you must select an xterm/tera-term window where one of active daemons is running, bring the daemon in a foreground execution by the Unix “fg” command if it is running in the background, and terminate it by typing “**control + c**”. This interrupted daemon immediately forwards the signal to all the other daemons to terminate themselves. Upon a normal termination, the daemon displays the following message:

```
^Cthrd: Interrupted.
  thrd: Closed all connections.
  thrd: Terminated.
```

An unexpected trap/interrupt or the Unix kill command may cause an abnormal termination and keep the previous socket connection in use. This may also cause several daemons to remain alive. When it happens accidentally, you are required to:

1. visit all the xterm windows where those daemons are still running
2. kill those daemons by typing “**control + c**”, and thereafter
3. wait for a few minutes until Unix recognizes that the previous socket port is already released, or edit your profile to change the port number.

### 3.4 Using Myrinet Zero-Copy Communication

The M++ daemon communicates with others using either the TCP/IP or the Myrinet GM library. The latter implements zero-copy communication in that so-called *pin-downed* pages are allocated independently of ordinary OS-managed pages, and are accessed directly by an underlying Myrinet network interface card. The M++ daemon has an option to schedule its threads in such *pin-downed* pages that the network interface card can transfer those threads without OS interventions. We have confirmed in [FSCK01] that this scheme improved migration performance 1.85 times better than using TCP/IP in a multi-agent application.
To implement inter-daemon communications with the GM library, use the commands listed below:

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Corresponding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daemon compilation</td>
<td>make gmw</td>
</tr>
<tr>
<td>Daemon invocation</td>
<td>thrd_gmw</td>
</tr>
<tr>
<td>Thread injection</td>
<td>inject_gmw hostname filename #threads arg1 arg2 arg3 ...</td>
</tr>
</tbody>
</table>

4 Writing M++ Threads

This section describes how to program M++ threads. You will first capture the overview of M++ executable model, thereafter understand how to define logical network components, and finally learn how to program M++ thread code.

Before going on to the following sub-sections, please be reminded of the usage of a pair of square brackets [ ] or [ ]:

- a pair of square brackets in italic [ ]
  Symbols inserted between a pair of italic brackets [ ] may be omitted.

- a pair of square brackets in bold [ ]
  this pair of brackets is a part of M++ statement or method.

4.1 Execution Model

M++ involves three levels of networks as shown in Figure 1. The lowest level is the physical network (Myrinet in both Tsukuba’s howdy and UWB’s medusa PC clusters, AP-NET in Ehime’s AP3000, and 100Mbps Ethernet in UCI’s iris Sparc cluster), which constitutes the underlying computational nodes. Superimposed on the physical layer is the daemon network, where each daemon is a Unix process executing and exchanging M++ threads with others. As described in Section 3.1, its processor mapping and system unique ID, termed daemon ID, are predefined in a user profile. The logical network is an application-specific computation network dynamically constructed by M++ threads on top of the daemon network. Each logical node has a node ID local to the corresponding daemon, while a logical link maintains a set of source and destination IDs, each local to the current and the destination logical node respectively.

M++ threads distinguish four C++ classes of network objects: daemon, daemonobj, node, and link. The daemon object provides M++ threads with the current daemon status. The daemonobj object stores user-defined data shared by all M++ threads
running on the same daemon process. The node object represents a logical network node, whose method and data members are accessed by M++ threads residing on this node. The link corresponds to a logical network link, makes its method and data members visible to threads residing on the both ends, (i.e., nodes) of this link. The daemon, node, and link objects also provide M++ threads with system-defined methods to inform of their network status: the current daemon ID, the total number of daemons, the host name, the node ID, the node class name, the number of threads on the current node, etc.. Details are given in Section 4.2. Sharing with these network objects is considered as a course of inter-thread communication, which however does not mean that M++ threads are provided with a complete view of distributed shared memory. Prior to accessing a given object, they must migrate themselves to it.

Figure 2 shows the M++ thread framework. A user can define various threads using the thread keyword. As in an ordinary C++ class, each thread allows its data members and member functions to be read and written in three access modes such as public, protected, and private. Their default attribute is private. The thread carries all data members with it whenever moving to a different node. However, it does not carry its member functions, which means that the same executable file must be made available
by NFS or manually copied in advance at each destination. The thread does not carry any local variables, either. This means that, while local variables including arguments can be still defined inside each member function, the validity of their contents is guaranteed only till a next thread migration.

The thread has constructors which can accept arguments. When a constructor is invoked with two arguments such as (int)argc and (char **)argv as shown in the following code, it can handle arguments passed from the inject command. In that case, the meaningful arguments start from argv[4]. (In precise, argv[0], argv[1], argv[2], and argv[3] include a string "inject", a target host name, this thread executable file, and # threads to be injected respectively.) The destructor has not yet been supported by the current version of M++. It will be made available in the following version.

The void main() function is the core of the thread, describing its autonomous behavior involved in network construction, navigation, inter-thread communication, and computation. It is automatically called by the system immediately after a constructor invocation. In Figure 2, a thread hops to a system-predefined node INIT on the daemon #0, and calls its work function. A return from the main() means the termination of the corresponding thread. Multiple inheritance and polymorphisms are available in the M++ thread. However, one exception is that main() cannot be inherited from base threads, (in other words, a sub-thread must define its own main().)

```cpp
thread SimpleThread {
    public: 
        SimpleThread( int argc, const char** argv ) : num( atoi(argv[4]) )
            { }
        void main() { hop( INIT @ 0 );
            work( );
        }
    private:
        void work() { cout << num << endl;
            int num;
        }
};
```

Figure 2: Thread Framework

Upon a startup, the system creates a logical node named INIT on every daemon. Any M++ thread may be injected (by the inject command from the shell or by another M++ thread) into any of the INIT nodes. It may then start network construction and migration using the following five groups of methods: [e]create, [e]destroy, [e]hop, [e]fork, and [e]inject. Note that the methods starting from ‘e’ return an error code, otherwise they return nothing for performance reasons.
• \textit{[e]}create node( ); permits an M++ thread to create a new logical node. See Section 4.2.3 for more details.

• \textit{[e]}create link( ); permits an M++ thread to create a new logical link. See Section 4.2.4 for more details.

• \textit{[e]}destroy node( ); permits an M++ thread to destroy an existing logical node. See Section 4.2.3 for more details.

• \textit{[e]}destroy link( ); permits an M++ thread to destroy an existing logical link. See Section 4.2.4 for more details.

• \textit{[e]}hop( ); navigates an M++ thread over an existing logical network. Its execution resumes from the statement following this hop keyword. See Section 4.3.6 for more details.

• \textit{[e]}fork( ); allows an M++ thread to spawn its copy on a specified destination and have it start from the statement following the fork method. The original thread continues its execution without waiting for the duplicated thread to be terminated. See Section 4.3.6 for more details.

• \textit{[e]}create thread( ); starts a different M++ thread from the beginning of its code on a specified destination. The original thread continues its execution without waiting for the injected thread to be terminated. See Section 4.3.2 for more details.

4.2 Network Components

This section gives the procedures to define and access the four M++ classes of network objects: \textit{daemon}, \textit{daemonobj}, \textit{node}, and \textit{link}. Note that, if those classes are defined in a header file, you must incorporate them into your \textit{mpp} source code by enumerating all those class names in a \textit{import} keyword. See Section 3.2 for the complete explanation.

4.2.1 The \textit{daemon} Class

M++ threads can obtain the current daemon status at any point of time by accessing the system-provided \textit{daemon} class object. This class includes the following public functions that can be accessed through the \textit{daemon} keyword followed by a dot "." and the corresponding function name.

• \textit{string name( );} returns the IP name of the current daemon.

• \textit{int id( );} returns the current daemon ID.
\* int total( ); returns the total number of daemons in the system.

4.2.2 The *daemonobj* Class

Unlike the system-provided *daemon* class, this is a user-provided class of which each M++ daemon instantiates only one object upon receiving the first M++ thread that has defined this *daemonobj* class. Thereafter, all M++ threads with the same *daemonobj* class definition can access this *daemonobj* instance as far as they reside on the same daemon. A user can define as many distinct *daemonobj* classes as needed, using the ordinary C++ class definition. However, each different M++ thread must declare which *daemonobj* class to use *a priori*. Such a declaration is achieved using the * daemonobj* keyword followed by a *daemonobj* class constructor call in front of the M++ thread definition. This is where the constructor receives its arguments which must be however literal values only.

Note that the classes defined for *daemonobjs* can be derived using single/multiple inheritance and from abstract classes as in C++. Anyway types of data members, method arguments, and pointers are permitted unless they references M++ threads which may migrate somewhere.

```cpp
class A { // The daemonobj A is declared as an ordinary C++ object.
public:
    A( int a, string b ) : { }
    ....;
};
d daemonobj A( 0, "abc" ); // A is the daemonobj which the thread T uses.
thread T {
    ....;
};
```

The *daemonobj* keyword is also used as the reference name of the *daemonobj* object. No matter what class a thread has declared as its *daemonobj*, it must use the reference name *daemonobj* in order to access the object. There are no other reference names than *daemonobj*. The following code example shows two different *daemonobj* class definitions such as *DaemonObj1* and *DaemonObj2*, each respectively used by the *Thread1* and *Thread2* M++ threads:

```cpp
class DaemonObj1 {
public:
    DaemonObj1( int num ) : num( num ) { }
    int num;
};
```
class DaemonObj2 {
public:
    DaemonObj1( const char* str ) { strcpy( DaemonObj2::str, str ); }
    char str[20];
};
daemonobj DaemonObj1( 0 );
thread Thread1 {
public:
    void main( ) { cout << 'I’m using DaemonObj1: ' << ++daemonobj.num << endl; }
};
daemonobj DaemonObj2( ‘Hello’ );
thread Thread2 {
public:
    void main( ) { cout << 'I’m using DaemonObj2: ' << daemonobj.str << endl; }
};

When receiving the first Thread1 thread, a daemon instantiates a DaemonObj1 object by invoking its constructor with 0 and resumes the Thread1 thread that thereafter increments the num variable to 1. The daemon however will no longer instantiate a new DaemonObj1 object but simply passes this reference to the second or the following Thread1 threads that visit this daemon. Similarly, the daemon instantiates a DaemonObj2 object with a string constant, “hello” when receiving the first Thread2 thread. Thus, all the Thread2 thread visiting this daemon retrieves “hello” as the content of daemonobj.str.

Note that each daemonobj objects, once allocated to a daemon, will never been deallocated until the system is terminated.

4.2.3 Nodes

The M++ node represents an application-specific logical network node. It is a place where M++ threads migrate themselves and carry out their task. A user can define various C++ classes for and instantiate them as M++ nodes. However, their actual instantiation takes place in response to runtime requests ordered by M++ threads. (This in turn means that no nodes are instantiated at compile time.) Once a node is instantiated from a user-defined C++ class, M++ threads may migrate to this node. All threads residing on the same node can share its data and method members. They are guaranteed to access the current node without intervened by any other threads unless they call either a migration or a synchronization method (see Section 4.3.7).

The following code shows an example of a class defined for nodes. Similar to classes used for daemonobj, it can be derived using single/multiple inheritance and from
abstract classes as in C++. Any types of variables and arguments are allowed except
pointers referencing M++ threads.

class Place {
public:
    Place( int num ) : num( num ) { }
    void setNum( int num ) { Place::num = num; }
    int getNum( ) { return num; }
    void printName( ) { cout << '"' << name() << '"' << endl; }
private:
    int num;
}

Note that this Place class includes the name( ) method whose body is not actually
specified inside this class definition. It is one of the system-predefined methods that
are available to any classes used for M++ nodes:

- int id( ); returns the current node ID.
- string name( ); returns the class name from which the current node was instan-
tiated.
- int thr_num( ); returns the total number of threads residing on the current node.

M++ threads are in charge of choosing which class it uses for a new node instantiation,
instantiating an object from the class, and mapping it onto a specific daemon as well
as giving a node ID to this object. To perform such a node creation, M++ threads
must call the following method in its main( ):

- [e]create node<NodeClassName>( args_list ) with ( NodeID[@DaemonID] );

where NodeClassName is the name of the class used for a node instantiation; aargs_list
is a list of arguments passed to this class constructor; NodeID is a daemon-unique
node identifier that must be a non-negative integer; and DaemonID is the identifier
of the daemon to which this node is mapped. If DaemonID is omitted, a new node
is created on the current daemon. This method returns mpp::Errno::noError on
success, otherwise mpp::Errno::createNodeFailed. If e is not prefixed to create, the
return value is always mpp::Errno::noError.

In order to access a node, M++ threads must move to the node by calling one of its
migration methods. One of such methods is:

- [e]hop( [NodeID[@DaemonID] );
where NodeID and DaemonID are the destination node’s and the destination daemon’s identifier respectively. Again, the omission of DaemonID specifies the current daemon as a default. This method returns mpp::Errno::noError on success, otherwise mpp::Errno::hopFailed. It always returns mpp::Errno::noError if it has no e prefix. To understand all thread migration methods in details, see Section 4.3.6.

M++ threads can access their current node through the reference name, node. There are no reference names other than node

- node.member: where member is one of system-provided node methods such as id( ), name( ), and thr_num( ).
- node<NodeClass>.member: where NodeClass is the class name from which this node was instantiated, and member is a member of this class.

M++ threads are also responsible to delete existing nodes. A node deletion is performed by:

- [e]destroy node( NodeID[@DaemonID] );

This method deletes a node with NodeID mapped on the daemon with DaemonID. (The omission of DaemonID means the current daemon.) It returns mpp::Errno::noError on success or always mpp::Errno::noError without the ‘e’ prefix, otherwise mpp::Errno::delete-NodeFailed. If ‘e’ is not prefixed, the return value is always mpp::Errno::noError. Note that if a node to be deleted has threads and links, it cannot be deleted. This also means that an M++ thread cannot delete the current node.

The following code example shows a sequence of node creation, access, and deletion:

class Place {
public:
    int num;
};

thread SimSpace {
public:
    void main( );
private:
    int i;
}

void SimSpace::main( ) {
    create node<Place> with( 0 );
    hop( 0 );
node<Place>::num = daemon.id( );
    hop( INIT );
    destroy node( 0 );
}

The SimSpace thread first creates a new node from the Place, gives the nodeID ‘0’ to this node, and maps it onto the current daemon. Thereafter, the thread migrates to this node and substitutes this node’s num variable with the current daemon id which is retrieved by daemon.id( ). Finally it hops back to the system-predefined INIT node and deletes the node whose id is 0.

4.2.4 Links

The M++ link represents a logical path along which M++ threads move from one node to another. While M++ threads can jump directly to a node addressed with its absolute node/daemon identifiers, links are another addressing option that permits threads to move to a node neighboring to the current node, (e.g., move to a left neighbor or a upward neighbor). Similar to the node, a user can define various C++ classes for and instantiate them as M++ links. Again, like nodes, the actual instantiation of links must be requested from and achieved by M++ threads at runtime. Once a link is instantiated from a user-defined C++ class, M++ threads may migrate along this link. A link is accessible from all threads residing on either one of the two end points of the same link, (i.e. one of the two nodes incident to the same link). Note that the system guarantees a non-interruptible access to a link. In other words, while a thread is executing a link method, no other threads can access any methods of the same link.

Each link has two identifies: source link ID and destination link ID. The former is unique to the source node, the latter unique to the destination node. Of notable is that the source and the destination IDs are recognized as the destination and the source IDs at the destination node. The link ID must be a non-negative integer.

The following code shows an example of a class defined for links. The C++ class inheritance is possible. Any types of variables and arguments are allowed except pointers referencing M++ threads.

class Path {
public:
    Path( int num ) : num( num ) { }
    void setName( int num ) { Place::num = num; }
    int getNum() { return num; }
    void printName() { cout << "I'm " << name( ) << endl; }
    void printId( ) { cout << "src=" << src_id( )

27
private:
   int num;
};

Note that this Path class includes name( ), src_id( ), and dest_id( ) whose body is not actually specified inside the class definition but implicitly provided by the system. Two groups of methods have been made available to M++ links: link-shared and link-local groups. The former provides general information shared among all links emanating from the same node, while the latter focuses on the information local to each link.

The following list summarizes link-shared methods:

- int count( ); returns the total number of links emanating from the current node.
- int min( ); returns the minimal link ID.
- int max( ); returns the maximal link ID.
- int next( int linkId ); returns the minimal link ID among those which are larger than linkId.
- int prev( int linkId ); returns the maximal link ID among those which are smaller than linkId.
- bool exists( int linkId ); returns true if there exists the link with linkId.

The following list summarizes link-local methods:

- int name( ); returns the class name from which the current link was instantiated from.
- int src_id( ); returns the source link ID.
- int dest_id( ); returns the destination link ID.
- int src_daemon_id( ); returns the source daemon ID.
- int dest_daemon_id( ); returns the destination daemon ID.
- int src_node_id( ); returns the source node ID.
- int dest_node_id( ); returns the destination node ID.

M++ threads are in charge of choosing which class it uses for a new link instantiation, instantiating an object from the class, and mapping it onto a specific daemon as well as giving a pair of source and destination link IDs to this object. To perform such a link creation, M++ threads must call the following method in its main( ):

28
• [e]create link<LinkClassName>( args_list ) with ( SrcID ) to ( NodeID[@DaemonID] ) with ( DestID );

where LinkClassName is the name of the class used for a link instantiation; args_list is a list of arguments passed to this class constructor; SrcID is a link ID unique to the source node; NodeID is the destination node identifier; DaemonID is the destination daemon’s identifier; and DestID is a link ID unique to the destination node. The omission of DaemonID means that the destination node is on the current daemon. This method returns mpp::Errno::noError on success, otherwise mpp::Errno::createLinkFailed. If ‘e’ is not prefixed to create, the return value is always mpp::Errno::noError.

Note that the source and the destination node of a link may be identical. In other words, the system permits threads to create a circle link departing from and arriving at the same node, even in which case the source and destination IDs of this link must be distinguished.

Once a link has been created, M++ threads can move to another node along the link by calling one of its migration methods. One of such methods is:

• [e]hopalong( srcLinkID );

where srcLinkID is the identifier of the link which emanates from the current node and along which the calling thread will move to its destination. This method returns mpp::Errno::noError on success, otherwise mpp::Errno::hopFailed. It always returns mpp::Errno::noError if it has no ‘e’ prefix. To understand all thread migration methods in details, see Section 4.3.6.

All links incident to the current node can be accessed through the reference name, link. There are no reference names other than link

• link.member: where member is one of link-shared methods which the system has predefined: count( ), min( ), max( ), next( ), prev( ), and exists( ).

• link[LinkID].member: where member is one of link-local methods which the system has predefined: name( ), src_id( ), dest_id( ), src_daemon_id( ), src_node_id( ), and dest_node_id( ).

• link[LinkID]<LinkClassName>.member: where LinkClassName is the class name from which this link was instantiated, and member is a member of this class.

M++ threads are also responsible to delete existing links. A link deletion is performed by:
• **[e]destroy link( LinkID );**

This method deletes a link whose source link ID is `LinkID`. It returns `mpp::Errno::noError` on success or always `mpp::Errno::noError` without the `e` prefix, otherwise `mpp::Errno::deleteLinkFailed`. If `‘e’` is not prefixed, the return value is always `mpp::Errno::noError`.

The following code example shows a sequence of link creation, access, and deletion:

```cpp
class Place { // a class used for node instantiation
};

class Path { // a class used for link instantiation
public:
    Path( ) : count( 0 ) { }
    int count;
}

thread SimSpace { // a thread instantiate a node and a link
public:
    void main( );
private:
    int i;
}

void SimSpace::main( ) { // a SimSpace thread's main behavior
    create node<Place> with( 0 );
    create link<Path> with( 0 ) to ( 0 ) with ( 1 );
    link[0]<Path>.count++;
    hopalong( 0 );
    cout << '‘link.count = ’' << link[0]<Path>.count++ << endl;
    hopalong( 1 );
    cout << '‘link.count = ’' << link[0]<Path>.count++ << endl;
    destroy link( 0 );
}
```

The `SimSpace` thread first instantiates a new `node` from the `Place`, gives the nodeID `‘0’` to this `node`, and maps it onto the current daemon. Thereafter, it establishes a new `link` from the current `node`, (i.e., an INIT `node` where injected) to the `node 0`. This `link` is accessible as `link 0` from the INIT `node` and as `link 1` from `node 0`. Thereafter, the thread increments this link’s `count` variable and hops it along to `node 0`. Again, it increments this link’s `count` variable and hops it along back to the INIT `node`. Finally, it deletes this link.
4.2.5 Classes Used for Network Components

It is possible to use the same class for definition of various network components such as daemonobj, node, and link, as far as it is an ordinary C++ class. However, if a class uses network-component-specific methods such as name and src_id, it must be used for representing only the network components that provide those specific methods at the system level. For instance, DaemonObj1 and DaemonObj2 in Section 4.2.2 are an ordinary C++ class which does not include any system-provided methods, and thus they can be also used as both nodes and links. On the other hand, Place in Section 4.2.3 includes a system-provided method, name( ) which is available for nodes and links, and therefore it must be used for nodes and links but not for daemonobjs.

4.3 Threads

M++ threads are active computation entities, capable of constructing and migrating over a logical network. They are defined using the thread keyword rather than class, however their definition takes a form similar to that of C++ classes except a few restrictions. Different types of M++ threads can be programmed inside the same m++ program.

Upon an instantiation, they are initialized with their constructor and automatically activated to execute their main method. Those steps are similar to Java threads: a constructor initialization and a run method invocation, however note that M++ threads run their main method automatically, while Java threads’ run method must be triggered by the start method.

The main method is the heart of M++ threads, and can perform:

- initialization of daemonobjs,
- creation of nodes, links, and other threads,
- access to all M++ objects such as daemons, daemonobjs, nodes, links, and other threads
- destruction of nodes, links and other threads,
- migration to another node,
- forking itself to the current or another node,
- direct communication with another thread, and
- inter-threads synchronization

Each of those manipulations listed here is explained in the following subsections.
4.3.1 Thread Definition

The definition of M++ thread starts from the thread keyword rather than class. It follows all C++ class rules for single/multiple inheritance, polymorphisms, and abstract definition, however there is one exception: the main( ) method defined inside each thread will not be inherited by any sub-threads. All sub-threads must define their own main( ) method.

If a thread definition has one or more pure virtual functions, it is regarded as an abstract thread and does not have to include its main( ) method. Nevertheless to say, no instantiation is possible from an abstract thread. It is used to derive sub-threads which no longer include pure virtual functions.

Data members declared in thread are carried with each individual thread upon a migration. However, there are three restrictions in the use of variables:

- Local variables are not carried with each thread. They can be still declared and used inside each method, however their contents are not guaranteed once a thread has migrated to another node.
- Arguments passed to any methods defined in thread are not carried with a thread, either. Their contents are not guaranteed once a thread has migrated to another node.
- Pointer contents are not guaranteed once a thread has migrated to a remote node, although they can be still defined as thread data members. This implies that memory allocated within thread using new must be freed with delete before a next migration. (Note that dynamic memory allocation works in any other classes, since they are immobile.)

The following code defines the Base thread from which the Sub thread is derived as a sub-thread.

```cpp
thread Base { // a super thread
public:
    Base( int num ) : num( num ) { }
protected:
    int getNum( ) { return( num ); }
    void setNum( int num ) { Base::num = num; }
    virtual void printNum( ) = 0;
private:
    int num;
};

thread Sub : public Base { // a sub thread
```

32
public:
    Sub( int argc, const char** argv ) : Base( atoi( argv[4] ) ) { }
    void main() { printNum( ); }
protected:
    void printNum( ) { cout << ‘‘Num = ‘‘ << getNum( ) << endl; }
};

The Base thread includes a pure virtual function, printNum( ), because of which it
does not have to include its own main( ) method. The Sub thread inherits Base
and implements the printNum( ) method. Since it have no more pure functions, it
must define the main( ) method which simply calls printNum( ). Its constructor
converts the argv[4] string into an integer that is passed to the Base constructor.
This integer value is stored in the num variable and printed out when the main( )
calls printNum( ).

4.3.2 Thread Creation and Constructor

M++ threads are instantiated either (1) by running the inject command from the
Unix shell or (2) executing the create statement from any other M++ thread.

1. inject:
The inject command creates one or more thread instances at a time. It takes the
following arguments:

    % inject hostname ThreadClassName #instances arg1 arg2 ..... 

where hostname is the IP name of a cluster computing node onto which you
will instantiate M++ threads; ThreadClassName is the name of thread class;
#threads is the number of instances created from ThreadClassName; and arg1
arg2 ... are the actual arguments passed to a constructor of each instance. All
instantiated threads start at the INIT node on the designated computing node.

2. create:
The create statement creates only one thread instance at a time. It takes one of
the following three forms:

• [ejcreate thread<ThreadClassName>( args_list ) to( NodeID[@DaemonID] )
];
where ThreadClassName is the name of thread class; args_list is a list of
char[] arguments passed to a constructor, (i.e., a list of character arrays); and
NodeID[@DaemonID] specifies a node where a thread is instantiated. If DaemonID
is omitted, the instance is placed at a designated node on the current daemon.
- `[e]create thread<ThreadClassName>( args_list ) along( LinkID );`
  where `ThreadClassName` is the name of thread class; `args_list` is a list of `char[]` arguments passed to a constructor, (i.e., a list of character arrays); and `LinkID` specifies a link whose destination node is where a new instance is created.

- `[e]create thread<ThreadClassName>( args_list )`
  where `ThreadClassName` is the name of thread class, and `args_list` is a list of `char[]` arguments passed to a constructor, (i.e., a list of character arrays). This statement instantiates a thread on the current node.

If the `create` statement has a prefix ‘e’, it returns `mpp::Errno::noError` on success, otherwise `mpp::Errno::injectFailed`. Without the ‘e’ prefix, it always returns `mpp::Errno::noError`.

Constructors of M++ thread may or may not accept arguments. If a thread is designed to always derive sub-threads, its constructors may accept any type of arguments. If a thread is however designed to accept arguments given from either `inject` or `create` upon its instantiation, it must include a constructor which receives `int argc` and `char** argv` arguments. The `argc` argument contains four plus the number of arguments passed from either `inject` or `create: arg1 arg2 ....` The `argv` arrays include:

<table>
<thead>
<tr>
<th>argv</th>
<th>when invoked from inject</th>
<th>when invoked from create</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>a string “inject”</td>
<td>null</td>
</tr>
<tr>
<td>argv[1]</td>
<td>the target host name where injected</td>
<td>null</td>
</tr>
<tr>
<td>argv[2]</td>
<td>this thread class name</td>
<td>null</td>
</tr>
<tr>
<td>argv[3]</td>
<td># of instances (represented in a string)</td>
<td>null</td>
</tr>
<tr>
<td>argv[4]</td>
<td>arg1</td>
<td>arg1</td>
</tr>
<tr>
<td>argv[5]</td>
<td>arg2</td>
<td>arg2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The following code defines two threads such as `Parent` and `Child`. `Parent` simply creates 10 `Child` instances on the current node, and passes two arguments to them. The first argument is “I am Thread No.” for all `Child` threads, whereas the second argument is “0”, “1”, “2”, ..., “9” to each of them. Each `Child` stores the first argument in the `msg` string variable, converts the second argument in the `id` integer variable, and prints out `msg` and `id` in its `main()` method.

```cpp
thread Child {
    public:
    Child( int argc, const char** argv ) : id( atoi( argv[5] ) ) {
        strncpy( msg, argv[4], 16 );
    }
    void main() { cout << msg << id << endl; }
}
```

34
private:
    char msg[16];
    int id;
}

thread Parent {
public:
    void main( );
private:
    int i;
    char numstr[10];
};

void Inject::main( ) {
    for ( i = 0; i < 10; i++ ) {
        printf( idstr, "\%d", i );
        create thread<PrintMsg>( "I am Thread No." , idstr );
    }
}

4.3.3 Thread Termination and Destructor

The M++ thread is terminated either implicitly by reaching the end of its main( ) method or explicitly by executing the following statement.

- terminate;
  terminates the invoking thread.

The current version of M++ has not supported the thread destructor. Although you may define it, nothing would happen.

4.3.4 Thread Status

Each M++ thread includes several system-provided methods that indicate the thread status. They are categorized into two groups: thread-shared and thread-local groups. The former provides general information shared among all threads residing on the same node, while the latter focuses on the information local to each thread.

The following list summarizes thread-shared methods:

- int count( ); returns the total number of threads residing on the current node.
- int min( ); returns the minimal thread ID.
• \textit{int max( )}; returns the maximal thread ID.

• \textit{int next( thread\_id threadId )}; returns the minimal thread ID among those which are larger than \textit{threadId}.

• \textit{int prev( thread\_id threadId )}; returns the maximal thread ID among those which are smaller than \textit{threadId}.

• \textit{bool exists( thread\_id threadId )}; returns true if there exists the thread with \textit{threadId}.

The following list summarizes \textit{thread}-local methods:

• \textit{int name( )}; returns the class name from which this \textit{thread} was instantiated.

• \textit{int src\_id( )}; returns the identifier of this \textit{thread}.

All \textit{threads} residing on the same \textit{node} can access one another through the reference name, \textit{thread}. There are no reference names other than \textit{thread}.

• \textit{thread.member}: where \textit{member} is one of thread-shared methods which the system has predefined: \textit{count( ), min( ), max( ), next( ), prev( ), and exists( ).}

• \textit{thread[ThreadID].member}: where \textit{member} is one of thread-local methods which the system has predefined: \textit{name( ) and id( ).}

• \textit{thread[ThreadID]<ThreadClass>.member}: where \textit{ThreadClass} is the class name from which this thread was instantiated, and \textit{member} is a member of this class.

\textbf{4.3.5 Network Construction}

The construction of logical networks is viewed as a series of creating and deleting \textit{nodes} and \textit{links}. Threads are only entities that can instantiate those network components.

• \textit{[e]create node< NodeClassName>( args\_list ) with ( NodeID[@DaemonID] )};
  Details are given in Section 4.2.3.

• \textit{[e]destroy node( NodeID[@DaemonID] )};
  Details are given in Section 4.2.3.

• \textit{[e]create link< LinkClassName>( args\_list ) with ( SrcID ) to ( NodeID[@DaemonID] ) with ( DestID )};
  Details are given in Section 4.2.4.
• 

\[ \text{destroy link( LinkID );} \]

Details are given in Section 4.2.4.

The following code permits an M++ thread to create a MyNode node on each daemon and establish a MyLink link to each of those nodes from the current working node, (i.e., the INIT node where it has been injected).

```cpp
class MyNode { } // a class used to instantiate a node
class MyLink { } // a class used to instantiate a link
thread StarNet { // a thread creating nodes and links
private:
    int i;
    int nodeId, linkId;
public:
    StarNet( int argc, char** argv ):
        nodeId( atoi( argv[4] ) ), linkId( atoi( argv[5] ) ) { }
    void main( ) {
        for ( i = 0; i < daemon::total(); i++ ) {
            create node<MyNode>( ) with ( nodeId @ i );
            create link<MyLink>( ) with ( i ) to ( nodeId @ i ) with ( linkId );
        }
    }
};
```

In this example, the StarNet thread receives two arguments, each stored into its nodeId and linkId variables. It creates a new node on each of all daemons whose identifier ranges from 0 to daemon::total(). Each node receives the content of nodeId as its identifier. Thus, all the node obtains the same identifier while each resides on a different daemon. As soon as the thread creates a new node, it establishes a new link to this node from the current node. The source link ID of each link is set to its destination daemon’s identifier, while the destination link ID is set to the content of linkId.

4.3.6 Thread Deployment

An M++ thread can migrate itself or deploy its copy over an existing logical network by calling one of the following methods. Similar to the network construction, if those methods are prefixed with 'e' and an error has occurred, they return an error code. Otherwise, they return mpp::errno::noError, (i.e., no errors.) For all those methods, if the DaemonId argument is omitted, their migration or deployment is bound for the local daemon. If the NodeId argument is omitted together, their destination is the current working node. Note that it is not permissible to omit only NodeId.
- `[e]hop( [NodeID[@DaemonID]] )`;
  migrates the calling thread to the node specified with NodeId and DaemonId. If this method is not prefixed with ‘e’ and an error has occurred at an migration, the calling thread is automatically terminated as displaying a termination message. If it is prefixed with ‘e’ and an error has occurred at an migration, the calling thread remains at the same node as receiving an error code.

- `[e]hopalong( SrcLinkID )`;
  migrates the calling thread along the link specified with SrcLinkId. If this method is not prefixed with ‘e’ and an error has occurred at an migration, the calling thread is automatically terminated as displaying a termination message. If it is prefixed with ‘e’ and an error has occurred, the calling thread remains at the same node as receiving an error code.

- `[e]fork( [NodeID[@DaemonID]] )`;
  deploys a child of the calling thread to the node specified with nodeId and daemonId. The child thread starts its execution from the statement next to `[e]fork( )`. The parent thread keeps running without waiting for its child.

- `[e]forkalong( SrcLinkID )`;
  deploys a child of the calling thread along the link specified with linkId. The child thread starts its execution from the statement next to `[e]forkalong( )`. The parent thread keeps running without waiting for its child.

The following code example permits an M++ thread to visit each leaf node of a star network already constructed by another M++ thread whose code was shown in Section 4.3.5.

```cpp
thread StarHopper {
  private:
    int i;
    int srcNode, srcDmn;
  public:
    StarHopper( int argc, char** argv ) { }
    void main( ) { }
    srcNode = node.id( ); // record the origin node id
    srcDmn = daemon.id( ); // record the origin daemon id
    for ( i = 0; i < daemon.total(); i++ ) { // for each daemon
      hopalong( i ); // hop along the link to it
      cout << "hello! at " << daemon.id( ) << endl; // display the dest. daemon id
      hop( srcNode@srcDmn ); // go back to the origin node
    }
  }
}
```
This example first memorizes the node and daemon identifiers of where a *StarHopper* thread has been injected. It thereafter repeats dispatching the thread along each different link emanating from the origin, labeled from 0 to `daemon.total( )` - 1, thus leading to each leaf daemon. Every time the thread visits on a different leaf node, it displays “hello!” and jumps back to the origin node.

4.3.7  Synchronization

The following statements are used for inter-threads synchronization.

- **sleep [node | daemon]**
  
  suspends the execution of the calling thread which will sleep until it is woken up by either a `wakeup` or a `wakeupall` statement.

- **wakeup ThreadID | node | daemon**
  
  wakes up the thread specified by `ThreadID` if it is given, wakes up one of the threads sleeping on the current node if the `node` parameter is given, or wakes up one of those sleeping on the current daemon if the `daemon` parameter is given. The woken-up thread will resume its execution after the point where it was suspended. A sleeping thread is chosen to wake up in a first-come-first-service order so that no starvation occurs.

- **wakeupall node | daemon**
  
  wakes up all threads sleeping on the current node or all those sleeping on the current daemon according to a parameter given to the `wakeupall` statement. All those threads will resume their execution after the point where they were suspended.

Non-interruptible execution is guaranteed by implementing every node as a monitor. Each node allows only one of the arriving threads to resume its execution and access the node variables. Thus, there is no concurrency inside a node, while threads residing at a different node may run concurrently. The daemon also guarantees non-interruptible access to each link. While a thread is executing a certain link’s method, no other threads can access the same link. The thread will suspend its non-interruptible execution and relinquish CPU to a next ready thread by calling one of the following methods

- **Network construction/destruction affecting a remote daemon**
  
  - `filecreate node` if it must be created on a remote daemon
  - `filedestroy node`
- [e]create link if it must be established to a remote daemon

- Link-accessing statements
  - `link[LinkID].member`
  - `link[LinkID]<LinkClass>.member`

- All thread deployment statements
  - [e]hop
  - [e]hopolong
  - [e]fork
  - [e]forkalong

- Inter-threads synchronization
  - `sleep`
  - `wakeup/wakeupall` if you have chosen `pthread` in your profile.

Note that an M++ thread never yields CPU unless it calls one of the above statements. Programmers are responsible to prevent a thread from hogging CPU, (i.e., to prevent the other threads from starving for it.) If an M++ thread needs to wait for another thread, it should either execute `sleep` to suspend its execution or repeatedly invoke `hop( )` to relinquish CPU to another thread. (The former is recommended for better performance.)

In the following code example, the `WakeUp` thread creates five `PrintMsg` threads and performs a busy wait until they increment the `num` node variable, display a “Good night” message, and fall asleep. Thereafter, the `WakeUp` thread prints out “Wake up”, and wakes up all those five threads that then reply “Good morning”.

```c++
const int threadNum = 5;

class Place { // a class used to instantiate a node
public:
    Place( ) : num( 0 ) {}  
    int num;
};

class PrintMsg { // a child thread created by a Wakeup thread
public:
    void main( ) {
        node<Place>.num++;
        cout << "Good night.\n" << endl;
        sleep node;
    }
};

class WakeUp { // a class containing the code of the thread
public:
    void main( ) {
        for (int i = 0; i < threadNum; i++) {
            node<PrintMsg> pm;
            pm.main();
            cout << "Wake up\n" << endl;
        }
    }
};

int main( ) {
    Place pl;
    WakeUp wu;
    wu.main();
    return 0;
}
```
cout << "'Good morning.'" << endl;
}
}

thread WakeUp { // a parent thread to spawn 5 PrintMsg threads
private:
    int i;
public:
    void main() {
    
    void WakeUp::main() {
        create node<Place> with( 0 );
        hop( 0 );

        for ( i = 0; i < threadNum ; i++ )
            create thread<PrintMsg>;

        while ( node<Place>.num != threadNum )
            hop( );

        cout << "'Wake up!!'" << endl;
        wakeupall node;
    }

    The above program runs in the following sequence of thread context switches:

    1. The WakeUp thread performs all thread instantiations without any intervention. Thereafter, it relinquishes CPU to one of the five PrintMsg threads by calling hop( ).

    2. Once acquiring CPU, each PrintMsg thread increments num, prints out a “Good night” message, and thereafter executes sleep that yields CPU to another thread. Eventually, all the PrintMsg threads have fallen asleep, and CPU comes back to the WakeUp thread.

    3. The WakeUp thread executes wakeupall node and terminates itself. Control goes to one of PrintMsg threads.

    4. The PrintMsg thread, upon displaying “Good night”, terminates its execution and passes control to another thread that is about to print out the same message.

4.3.8 Node Clocking

This feature permits an M++ thread to invoke the same method of all nodes existing over the system. The method to be defined in node classes must have the following
interface:

void clocking( int time );

To invoke all the clocking methods, an M++ thread must execute the following clocking statement in its main( ) method.

- clocking time;
  passes an integer argument, time to the clocking method of all nodes.

With node clocking, a thread can avoid visiting all nodes over the system in order to invoke their clocking method, which may be useful for simulation applications to tick their virtual time.

Since a thread can access links while executing a method of the current node, (i.e., it can call link methods from a node method), the clocking method can be used to exchange data via links between any two neighboring nodes at a time.

Note that node clocking does not ideally mean a simultaneous invocation of all node’s clocking( ) method. In particular, clocking( ) methods on the same daemon will be invoked in sequential. Of course, such calling sequence is not deterministic, either.

4.3.9 Communication

While threads share node variables on the same node and thus use them as indirect communication media, they can also establish direct communication by calling each other’s public method. Note that a called thread must be sleeping on the same node where the calling node resides.

- thread[ ThreadID].member
  calls a predefined member function, (i.e., name( ) or id( )) of the thread sleeping on the current node and specified in ThreadID.

- thread[ ThreadId]< ThreadClass >.member
  calls a user-defined member function of the thread sleeping on the current node and specified in ThreadID.

The following program includes two threads such as Container and Reader. The former receives one argument upon its initialization and maintains it as an integer value. The latter, through the inter-thread communication, reads this integer value of each Container thread residing on the current node. More specifically, Reader scans
all threads whose identifications can be found from thread.min( ) to thread.max( ),
checks if each of them has "Container" as its class name, and, if so, calls its getData( )
method to retrieve the number value.
	hread Container { // a thread keeping an integer value which will be read by Reader
public:
    Container( int argc, const char** argv ) : number( atoi( argv[4] ) ) { }
    int getData( ) { return number; }
    void main( ) { sleep; }
private:
    int number;
};

thread Reader { // a thread reading each Container's integer value
public:
    void main( );
private:
    thread_id i;
}

void Reader::main( ) {
    if ( thread.count( ) > 0 ) {
        for ( i = thread.min( ) ; i <= thread.max( ) ; i = thread.next( i ) )
            if ( thread[i].name( ) == "Container" )
                cout << "Container[" << i << "]'s data = ",
                    << thread[i]<Container>.getData( ) << endl;
    }
}

4.3.10 Global Virtual Time (Soon Made Available)

This feature has been supported at the daemon level but not yet at the language
level. It will be soon made available.

Using the M++ global virtual time (abbreviated as GVT), threads can schedule their
next action at a certain logical time that starts from time 0. This feature is intended
to ease discrete-event simulations with M++. The following five methods are used
to start, schedule thread actions along, and stop GVT.

- void gvt_ready( )
A thread which wants to participate in a GVT computation must call gvt_ready in
prior to calling any of the following methods: [e]hop, [e]fork, [e]create, [e]destroy
and all GVT methods except gvt_start.

When calling gvt_ready at time 0, the thread waits until another thread calls
gvt_start that triggers time ticking. If gvt_start has been already called and a
GVT computation is in progress, the thread does not suspend its execution and goes forward to the next \textit{gvt\_delta} function call.

- \textbf{void gvt\_start( )}
  triggers GVT ticking from 0 and wakes up all threads which have called \textit{gvt\_ready}. A thread calling \textit{gvt\_start} implicitly participates in a GVT computation and thus does not have to call \textit{gvt\_ready}.

- \textbf{void gvt\_delta( int dTime )}
  Once participating in a GVT computation through \textit{gvt\_ready} or \textit{gvt\_start}, a thread may call \textit{gvt\_delta} to suspend its execution until GVT reaches the current time plus \textit{dTime}.

- \textbf{int gvt\_time( )}
  returns the current virtual time which is 0 or a positive integer.

- \textbf{void gvt\_end( )}
  gets out of the current GVT computation. Each participating thread implicitly leaves the GVT computation upon its termination. However, if a thread needs to behave regardless of GVT, it must explicitly call \textit{gvt\_end}.

Note that GVT does not tick up as far as any participating threads are still active. Once all of them have called \textit{gvt\_delta} to suspend themselves, the system increments GVT to the nearest future time at which at least one thread schedules its next action.

5 Examples

This section deals with two entire sets of M++ code such as Section 5.1: Gas Station and Section 5.2: Ant Farm. The former is a uniprocessor example program in that \textit{SportsCar} and \textit{Truck} agents stop by a gas station to be pumped by a \textit{GSCashier} agent. The latter is a multiprocessor application to find the gene of the most efficient ant forager by walking ants over a meshed farm constructed over multiple workstations.

5.1 Gas Station

This simple program models a gas station cashier fueling two different typed vehicles such as sports-cars and trucks. As shown in Figure 3, the former receives regular oil and the latter diesel.

Those three simulation entities are programmed as a \textit{Cashier}, a \textit{SportsCar}, and a \textit{Truck} thread respectively. In particular, the latter two threads are derived from the
Car base thread. In addition to those threads, the program defines the GasStation and the Fuel class, each instantiated as where Cashier works and what it serves.

5.1.1 GasStation/Fuel Classes

The gasStationId constant integer defines the id of the node where a Cashier thread works. It simply specifies 0.

The GasStation class includes nothing. It is used for a Cashier thread to instantiate its working node (whose id is gasStationId, i.e., 0).

The Fuel class includes two private integer variables: dieselOilId and regularOil, each having 1000 (gallons). The class provides four public methods:
<table>
<thead>
<tr>
<th>Methods</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int useDieselOil( amount )</td>
<td>subtract amount or all from diesel and return this amount</td>
</tr>
<tr>
<td>int useRegularOil( amount )</td>
<td>subtract amount or all from regular and return this amount</td>
</tr>
<tr>
<td>void print( )</td>
<td>display the amount of diesel/regular oil left</td>
</tr>
<tr>
<td>bool empty( )</td>
<td>return true if both diesel and regular are empty</td>
</tr>
</tbody>
</table>

```cpp
const int gasStationId( 0 );    // node id
class GasStation               // node class
{
};
class Fuel                    // maintained by GSCashier
{
public:
    Fuel( ) : dieselOil( 1000 ), regularOil( 1000 ) {}
    int useDieselOil( int amount ) // subtract amount if possible, otherwise
        { // subtract all from diesel diesel oil
        if( dieselOil < amount )
            amount = dieselOil;
        dieselOil -= amount;
        return( amount );
    }
    int useRegularOil( int amount ) // subtract amount if possible, otherwise
        { // subtract all from regular oil
        if( regularOil < amount )
            amount = regularOil;
        regularOil -= amount;
        return( amount );
    }
    void print()                   // print out the amount of diesel/regular
        { cout << "Diesel: " << dieselOil << " Regular: " << regularOil << endl; }
    bool empty()                   // true if no oil remains.
        { return( dieselOil == 0 && regularOil == 0 ); }
private:
    int dieselOil;
    int regularOil;
};
```
5.1.2 Car/SportsCar/Truck Threads

The Car thread is the base of both SportsCar and Truck threads. It includes two integers: fuel indicating the fuel remaining in its tank and tankmax specifying the tank capacity. Both are initialized upon a constructor invocation. This base thread has two public methods:

<table>
<thead>
<tr>
<th>Methods</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>void injectFuel( Fuel&amp; fuel)</td>
<td>pure virtual function implemented by SportsCar and Truck</td>
</tr>
<tr>
<td>void receiveFuel( )</td>
<td>wake up and wait for a GSCashier thread that pumps oil and signals it back</td>
</tr>
</tbody>
</table>

Note that, since Car is an abstract thread, it does not have the main( ) method.

```cpp
thread Car
{
public:
    Car( int tankMax ) : tankMax( tankMax ), fuel( 0 ){}
    virtual void injectFuel( Fuel& fuel ) = 0; // implemented by each sub thread
protected:
    void receiveFuel() // find and wait for GSCashier to complete pumping gas
    {
        if( thread.count() )
            for( i = thread.min(); i <= thread.max(); i = thread.next( i ) )
                if( thread[i].name() == "GSCashier" )
                    {  
                        wakeup i; // wake up GSCashier
                        sleep; // sleep till GSCashier pump fuel and wake me up
                        break;
                    }
    }

    int fuel;     // the current amount of fuel stored in the tank
    int tankMax;  // the tank capacity
    thread_id i;  // the id of GSCashier thread
};
```

The Truck thread implements the injectFuel method that actually receives a Fuel object from GSCashier and calls its useDieselOil( ) method. Through useDieselOil, the Truck thread fills its diesel tank.

The main( ) method, (i.e., Truck’s behavioral scenario), migrates Truck to where GSCashier is working; wakes up and waits for GSCashier by calling receiveFuel( ) method; and resumes its execution after GSCashier calls injectFuel( ) to pump fuel and notify it.
thread Truck : public Car
{
public:
Truck( int argc, const char** argv ) : Car( 100 ) {}
void injectFuel(Fuel& fuel) // called by GSCashier
{ Car::fuel = fuel.useDieselOil( tankMax - Car::fuel ); }
void main()
{
    hop( gasStationId );
cout << "Truck: I'm at the GasStation." << endl;
receiveFuel(); // wait till GSCashier pumps fuel
if( fuel == tankMax )
cout << "Truck: I'm filled with fuel." << endl;
}
};

The SportsCar thread implements the injectFuel method that actually receives a Fuel object from GSCashier and calls its useRegularOil( ) method. Through useRegularOil, the SportsCar thread fills its regular-oil tank.

Its main( ) method is the same as that of Truck except some message-printing statements.

thread SportsCar : public Car
{
public:
SportsCar( int argc, const char** argv ) : Car( 50 ) {}
void injectFuel(Fuel& fuel)
{ Car::fuel = fuel.useRegularOil( tankMax - Car::fuel ); }
void main()
{
    hop( gasStationId );
cout << "SportsCar: I'm at the GasStation." << endl;
receiveFuel();
if( fuel == tankMax )
cout << "SportsCar: I'm filled with fuel." << endl;
}
};

5.1.3 GSCashier

The GSCashier thread creates a new node with its gasStationId (= zero), hops to it, and repeats the rest of its behavior enclosed in its while loop. It sleeps until woken by either Truck or SportsCar. Thereafter, it identifies which vehicle thread has woken it up and calls this vehicle's injectFuel method, as passing its Fuel object, which pumps
the appropriate type of oil to the vehicle. Finally, GSCashier wakes up the vehicle thread and displays its fuel information.

```cpp
thread GSCashier
{
  public:
  void main()
  {
    create node<GasStation> with( gasStationId ); // create a working place
    hop( gasStationId ); // work there

    while( 1 )
    {
      cout << "GSCashier: Waiting for cars." << endl;
      sleep; // wait till a new vehicle comes around

      if( thread.count() )
        for( i = thread.min(); i <= thread.max(); i = thread.next( i ) )
        {
          thread[i]<Car>.injectFuel( fuel );
          cout << "GSCashier: Injected fuel to "
               << thread[i].name() << endl;
          wakeup i;

          fuel.print();
          if( fuel.empty() )
            cout << "Fuel is empty." << endl;
        }
    }
  }
private:
  thread_id i;
  Fuel fuel;
};
```

All the source code was programmed in `/home/m++/appl/m++/Car/Car.mpp`. To compile and execute it, follow the instruction given in Section 3.2.

### 5.2 Ant Farm

As shown in Figure 4, this multiprocessor application finds the most efficient ant forager by distributing food on a meshed network and walking many ant threads, each carrying different behavioral gene. To perform such a simulation, the program includes the following five types of M++ threads:

1. **Mesh** constructs a given size of mesh network.
2. *InitEnvironment* distributes food on four cells, each located at diagonally one inside a different corner of the mesh network.

3. *Ant* searches for and bring back food to the nest.

4. *Got* performs inter-threads synchronization every tick of simulation time.

5. *CheckResult* shows food, pheromone and ant distribution over the network after simulation.

The following subsections list each of those thread programs.

### 5.2.1 DaemonVar/Node/Link Definitions

Those classes are used to construct a *farm* simulation space where *ant* agents walk around.

**DaemonVar**

This is a *daemonobj* class mainly used for two functionalities such as virtual time maintenance and node-to-daemon mapping. Note that we developed this ant farm application before completing the *M++* global virtual time maintenance feature that therefore had to be implemented at an application level. The following summarizes the variables defined in *DaemonVar*. 

50
<table>
<thead>
<tr>
<th>type</th>
<th>name</th>
<th>used for</th>
<th>descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>daemonTotal</td>
<td>node mapping</td>
<td># of daemons over the system</td>
</tr>
<tr>
<td>int</td>
<td>meshsize</td>
<td>node mapping</td>
<td>a meshsize² farm created</td>
</tr>
<tr>
<td>int</td>
<td>divider</td>
<td>node mapping</td>
<td>dividing farm in 1: strip or 0: lattice</td>
</tr>
<tr>
<td>int</td>
<td>ant_num</td>
<td>time maintenance</td>
<td># of ants on this daemon</td>
</tr>
<tr>
<td>int</td>
<td>live_ant</td>
<td>time maintenance</td>
<td># of active ants on this daemon</td>
</tr>
<tr>
<td>int</td>
<td>sleep_ant</td>
<td>time maintenance</td>
<td># of sleeping ants on this daemon</td>
</tr>
<tr>
<td>int</td>
<td>gvt</td>
<td>time maintenance</td>
<td>the current global virtual time</td>
</tr>
<tr>
<td>bool</td>
<td>is_reported</td>
<td>time maintenance</td>
<td>false if none of daemons has informed the master of ant info.</td>
</tr>
<tr>
<td>bool</td>
<td>gvt_kicker_flag</td>
<td>time maintenance</td>
<td>false if this daemon has not informed the master of ant info.</td>
</tr>
<tr>
<td>Timer</td>
<td>timer</td>
<td>performance</td>
<td>see section ??</td>
</tr>
</tbody>
</table>

```cpp
import <cmath> {}
import <cstdlib> {}
import "Timer.h" { class Timer; } // used for performance evaluation

const bool DEBUG( false );

class DaemonVar
{
public:
  DaemonVar() {
    ant_num = 0;
    live_ant = 0;
    sleep_ant = 0;
    gvt = 0;
    is_reported = false;
    gvt_kicker_flag = false;
    divider = 0;
  }
  int daemonTotal; // # daemons over the system
  int meshsize; // a meshsize x meshsize farm created
  int divider; // dividing farm in 1: stripe or 0: lattice
  int ant_num; // # ants residing on this daemon
  int live_ant; // # active ants on this daemon
  int sleep_ant; // # sleeping ants on this daemon
  int gvt; // the current virtual time
  bool is_reported; // false if none of daemons informed master of ants
  bool gvt_kicker_flag; // false if a daemon informed master of ants info.
  Timer timer; // used for performance evaluation

  int getNodeID( int x ,int y ) { // decide a node id from [x,y] coordinates
    return y * meshsize + x;
  }
  int getDaemonID( int nodeId ) { // decide a daemon id from a given node id
    int dId;

    dId = 0;
    if ( divider == 1 ) // stripe mapping
```
dId = ((nodeId * daemonTotal) /(meshsize*meshsize));
else { // lattice mapping
  switch( daemonTotal ) {
    case 1:
      dId = 0;
      break;
    case 2:
      if ( nodeId < meshsize * meshsize / 2 )
        dId = 0;
      else
        dId = 1;
      break;
    case 4:
      if ( nodeId < meshsize * meshsize / 2 ) {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 0;
        else
          dId = 1;
      } else {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 2;
        else
          dId = 3;
      }
      break;
    case 8:
      if ( nodeId < meshsize * meshsize / 4 ) {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 0;
        else
          dId = 1;
      } else if ( nodeId < meshsize * meshsize / 2 ) {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 2;
        else
          dId = 3;
      } else if ( nodeId < meshsize * meshsize / 4 * 3 ) {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 4;
        else
          dId = 5;
      } else {
        if ( nodeId % meshsize < meshsize / 2 )
          dId = 6;
        else
          dId = 7;
      }
    }
break;
default:
dId = 0;
break;
}
}
return dId;
};

Trail
This is a node class that maintains geometric information and food. Each node is first created by a Mesh thread, is thereafter initialized by a InitEnvironment, is used for ant threads to communicate with one another, and is finally checked of its data by a CheckResult thread.

class Trail {
public:

/***************
//Class Variable
***************
int x, y; // coordinates
int nest; // 1 if it is the nest, otherwise 0
int food; // the amount of food left on this node
int ant_num; // # of ants on this node
int pheronom_to_food; // the level of food-directive pheromone left on this node
int pheronom_to_nest; // the level of nest-directive pheromone left on this node

/***************
//Method
***************
Trail(int x, int y): x(x), y(y){ // constructor
nest = 0;
food = 0;
ant_num = 0;
pheronom_to_food = 0;
pheronom_to_nest = 0;
}

void drop_pheromon_to_food(int ph){ // increment food-directive pheromone
pheronom_to_food = pheronom_to_food + ph;
}

void drop_pheromon_to_nest(int ph){ // increment nest-directive pheromone
pheronom_to_nest = pheronom_to_nest + ph;
}

53
int get_pheromone_to_food() {   // detect food-directive pheromone
    return pheromone_to_food ;
}

int get_pheromone_to_nest(){  // detect nest-directive pheromone
    return pheromone_to_nest ;
}

void ant_in(){                     // one ant came in this node
    ant_num++;
}

void ant_out(){                     // one ant left this node
    ant_num--;
}

int is_food(){                      // return the amount of food
    return (food > 0);
}

int get_food(int f){              // return a requested or all
    if (food < f) // remaining food
        f = food;
    food = food - f;
    return f;
}

void set_food(int f){             // initialize the amount of food
    food=f;
}

void drop_food(int f){            // receive food dropped by an ant
    food=food + f;
}

int is_nest(){                     // 1 if this is the nest
    return nest;
}

Way
This is used to instantiate links. The current implementation does not use any methods defined in Way.

class Way {
public:
    /***************
    // Class Variable
***************
int i;
int pheromon_to_food;
int pheromon_to_nest;

***************
// Method
***************
Way(int i):i(i){}

/* Ant threads call none of those functions in the current implementation */
void drop_pheromone_to_food(int ph){
   pheromon_to_food = pheromon_to_food + ph;
}

void drop_pheromone_to_nest(int ph){
   pheromon_to_nest = pheromon_to_nest + ph;
}

int get_pheromon_to_food(){
   return pheromon_to_food;
}

int get_pheromon_to_nest(){
   return pheromon_to_nest;
}
);

5.2.2 Mesh Thread

The Mesh thread obtains a mesh size and a node mapping scheme from the first and
the second argument respectively. For instance, given 200 as the mesh size and 1 as
the node mapping scheme, it creates a $200 \times 200$ mesh network, divides it in stripe,
and maps each divided space to a different processor. If a user wants to divide the
network in lattice, the second argument must be set to 0.

The Mesh thread carries following six thread variables with it:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>meshsize</td>
<td>a mesh size</td>
</tr>
<tr>
<td>divider</td>
<td>a node mapping scheme (0 = lattice, 1 = stripe)</td>
</tr>
<tr>
<td>i,j</td>
<td>coordinates of each Trail node</td>
</tr>
<tr>
<td>daemonNO</td>
<td>the daemon ID computed from a given node ID</td>
</tr>
<tr>
<td>ToNodeID</td>
<td>the node ID computed from given i,j coordinates</td>
</tr>
</tbody>
</table>

In addition, the mesh thread has the following two methods:
• createMesh first creates node[i,j] where 0 <= i < meshsize and 0 <= j < meshsize, and then call its link_create method to establishes links emanating from each node.

• link_create creates a link from a given node to each of 8 neighboring nodes.

The Mesh thread is the first one injected into the system, which in turn means that Mesh is in charge of initializing all daemonobj objects. The following lists a complete code of the Mesh thread.

daemonobj DaemonVar();  // initialize daemonobj at each daemon

thread Mesh
{
public:
    Mesh( int argc, const char** argv ) :
    void main();  // argv[4]: meshsize, argv[5]: divider
private:
    int meshsize;  // a mesh size
    int i,j;  // coordinates of each Trail node
    int daemonN0;  // the daemon ID computed from a given node ID
    int ToNodeID;  // the node ID computed from given i,j coordinates
    int divider;  // a node mapping scheme (0=lattice, 1=strips)

    void link_create();
    void createMesh();
};

void Mesh::link_create()  // creates a link to each of 8 neighbors.
{
    // if not on y axis, create a link to west(0)
    if( node<Trail>.x != 0 && node<Trail>.y != meshsize - 1 ){
        ToNodeID = node.id() - meshsize;
        create link<Way>( 0 ) with( 0 )
        to( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) ) with( 4 );
    }
    // if not on y axis nor on the upper boundary, create a link to northwest(1)
    if( node<Trail>.x != 0 &

56
} // if not on the upper nor the left boundary, create a link to northeast(3)
if( node<Trail>.y != meshsize - 1 & & node<Trail>.x != meshsize - 1 ){
    ToNodeID = node.id() + meshsize + 1;
    create link<Way>( 0 ) with( 3 )
    to( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) ) with( 7 );
}
}

void Mesh::createMesh() // create a [i,j] node and establish links from it.
{
    for(i=0;i<meshsize;i++){
        for(j=0;j<meshsize;j++){
            ToNodeID = meshsize * i+j; // compute node id from i,j coordinates
            create node<Trail>( i, j ) // create a node
            with( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
        }
    }
}

for(i=0;i<meshsize;i++){
    for(j=0;j<meshsize;j++){
        ToNodeID = i*meshsize+j;
        hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) ); // hop to each node
        link_create(); // create links from it
    }
}

void Mesh::main()
{
    for( i = 0 ; i < daemon.total() ; i++ ) {
        hop( INIT@i ); // visit each daemon
        daemonobj.daemonTotal = daemon.total(); // initialize daemonobj
        daemonobj.meshsize = meshsize;
        daemonobj.divider = divider;
    }
    createMesh(); // create a mesh network
    for ( i = 0 ; i < daemon.total() ; i++ ) { // revisit each daemon
        hop( INIT@i ); // print out its task completion
        cout << "daemonobj.meshsize = " << daemonobj.meshsize << endl;
        cout << "daemonobj.divider = " << daemonobj.divider << endl;
    }
}
5.2.3 *InitEnvironment* Thread

The *InitEnvironment* thread receives one argument as the number of ants to inject, records it in each daemon’s *daemonobj.ant_num*, and distributes food on four cells, each located at diagonally one inside a different corner of the mesh network. Given $N$ as the mesh size, the cells to include food are: $cell[1,1]$, $cell[1,N-2]$, $cell[N-2,1]$, and $cell[N-2,N-2]$.

The *InitEnvironment* thread carries following three thread variables with it:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>a for-loop control variable</td>
</tr>
<tr>
<td>ToNodeID</td>
<td>the node ID computed from given $i,j$ coordinates</td>
</tr>
<tr>
<td>ant_num</td>
<td>a variable to store the total number of ants</td>
</tr>
</tbody>
</table>

The *InitEnvironment* thread initializes the simulation space using its three methods below:

- *setNest* sets the central cell of the network to the nest.
- *setFood* places 10,000 food on four cells, each located at diagonally one inside each corner of the mesh network
- *set_daemon_variable* have each daemon record the number of ants to be injected.

The following lists a complete code of the *InitEnvironment* thread:

```cpp
thread InitEnvironment
{
public:
    InitEnvironment( int argc, const char** argv ) :     
        ant_num( atoi( argv[4] ) ) {} // argv[4]: #ants to be injected
    void main();
    void setNest();
    void setFood();
    void set_daemon_variable();
private:
    int k; // a for-iteration variable
    int ToNodeID; // a node id computed from i,j
    int ant_num; // #ants to be injected
};

void InitEnvironment::setNest() // set the central cell to the next
{
    int halfsize = daemonobj.meshsize / 2;
}
```

58
ToNodeID = halfsize * daemonobj.meshsize + halfsize;
hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
node<Trail>.nest = 1;

printf( "NEST LOCATION: (%d,%d)\n", node<Trail>.x, node<Trail>.y );
printf( "IS NEST: %d\n", node<Trail>.is_nest() );
}

void InitEnvironment::setFood() // place 1000 food each at four nodes
{
    //**** North East
    ToNodeID = daemonobj.meshsize + (daemonobj.meshsize-1);
hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
node<Trail>.food=10000;

    //**** South East
    ToNodeID = (daemonobj.meshsize-1)*daemonobj.meshsize + (daemonobj.meshsize-1);
hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
node<Trail>.food=10000;

    //**** North West
    ToNodeID = daemonobj.meshsize + 1;
hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
node<Trail>.food=10000;

    //**** South West
    ToNodeID = (daemonobj.meshsize-1 )*daemonobj.meshsize + 1;
hop( ToNodeID @ daemonobj.getDaemonID( ToNodeID ) );
node<Trail>.food=10000;
}

void InitEnvironment::set_daemon_variable()
{
    for ( k = 0; k < daemonobj.daemonTotal; ++k ){
        hop( INIT @ k ); // visit each daemon
daemonobj.ant_num = ant_num; // # ants to be injected
daemonobj.live_ant = ant_num; // they are initially all active
daemonobj.sleep_ant = 0; // they are not sleeping of course
daemonobj.gvt = 0; // simulation starts from time 0
daemonobj.is_reported = false; // 1st synchronization has not been reported

        if( DEBUG )
        {
            printf("--------set_daemon_variable--------\n");
            printf("daemonobj.meshsize:%d\n",daemonobj.meshsize );
            printf("daemonobj.ant_num:%d\n",daemonobj.ant_num );
            printf("daemonobj.live_ant :%d\n", daemonobj.live_ant);
        }
    }
}
printf("daemonobj.sleep_ant:%d\n",daemonobj.sleep_ant );
printf("daemonobj.gvt :%d\n", daemonobj.gvt);
}
}

void InitEnvironment::main()
{
    cout << "daemonobj.meshsize = " << daemonobj.meshsize << endl;
    cout << "daemonobj.divider = " << daemonobj.divider << endl;

    setNest();
    setFood();
    setdaemon_variable();

    cout <<"init environment END " <<endl;
}

5.2.4 Ant Thread

The Ant thread repeats the following actions every simulation cycle: (1) pick up food if it finds any, (2) drop off food in the nest or on route to the nest accidentally, (3) follow the strongest pheromone, (4) decide the next direction if it feels no pheromone, and (5) secrete some pheromone and walk to the next node. To give more reality to each action, the gene of each ant characterizes its average number of food to pick up, average pheromone amount to emit, pheromone sensitivity, and durability to walk straight. Those characteristic parameters are maintained in the following variables:
Variables | Descriptions
---|---
ToNodeID | the node ID computed from given i,j coordinates
food | the amount of food which this ant is currently holding
total_food | the total amount of food which this ant has carried to the nest
step | the current simulation time
direction | the next moving direction
ph_react | the current threshold of pheromone amount for ant to react.
i, j | for-loop control variables
border | a flag to indicate if the ant is on a processor border
randVar | the latest random number obtained from mand_r( )
gene | gene information obtained using random( )
norm_u | the average of this ant's pheromone reaction level
ph_react0 | the initial pheromone reaction threshold
pois_l | the average amount of food this ant picks up
keep_p | the probability to keep holding the food which the ant picked up
straight_p | the probability to go straight
trust_p | the probability to trust the latest pheromone information
seed | this ant's seed used for rand_r( )
temp_node_id | the node id to make the ant return to this node later
temp_daemon_id | the daemon id to make the ant return to this daemon later
activate_pheromone | the flag to make the ant follow pheromone
temp_max | the variable to store pheromone considered maximal temporarily
temp_direction | the variable to store the next moving direction temporarily

The following methods describe each action for an Ant to take:

- **init** calls set_gene to initialize thread variables based on a given gene and migrates a thread to the nest node.

- **set_gene** randomly decides the genetic code of each ant and initializes thread variables such as norm_u, ph_react0, pois_l, keep_p, straight_p, trust_p, and seed.

- **synchronize** migrates this ant to the INIT node on daemon #0 if it is the first one to do so, injects a new got thread if it has not yet been injected, comes back to the original node, and sleeps for the got thread to wake up this ant.

- **init_at_trail** increments the number of ants residing on this node and adds 10 to its own ph.react value, (which makes this ant less sensitive to pheromone.

- **work_at_trail** gets food if the current node is not the nest, drops food if it is the nest, and accidentally drops off food on route to the nest.

- **feelpheromone** replaces this ant’s variables tempMax and tempDir with the maximal pheromone and its direction which this ant has temporarily detected.

- **decide_direction** investigates the maximal pheromone and its direction by walking through all the cells neighbors of the current cell and decides the next moving direction based on this investigation, if this ant trusts this pheromone information. If not, the ant randomly chooses the next direction to walk.
• go_next_trail drops its pheromone, decrements the number of ants residing on the current node, and actually migrates the ant to the next cell.

• fact computes the factorial of a given argument.

• normal computes the normal distribution, given a average parameter and a random value.

• poisson computes the poisson distribution, given a lambda parameter and a random value.

The following shows a complete code of the Ant thread:

thread Gvt; // Ant spawns a Gvt thread
const int LIFE_STEP( 4001 ); // simulation ends in time 4001
extern "C" int rand_r(unsigned int *seed); // rand_r is used.

thread Ant
{
    public:
        Ant( int argc, const char** argv ) :
            activate_pheromone( ( atoi( argv[4] ) == 1 ) ? true : false ) {} 
            void main(); // if 1st arg is 1, pheromone is used.

    private:
        int ToNodeID; // the node ID computed from given i,j coordinates
        int food; // the amount of food which this ant is holding
        int total_food; // the total amount of food which this ant carried to the nest
        int step; // the current simulation time
        int direction; // the next moving direction
        int ph_react; // the current threshold of pheromone amount for ant to react
        int i, j; // for-loop control variables
        bool border; // a flag to indicate if the ant is on a processor border

        //GENE
        short gene; // gene information obtained using random( )
        double norm_u; // the ave. of this ant’s pheromone reaction level
        int ph_react0; // the initial pheromone reaction threshold
        double pois_1; // the ave. amount of food which this ant picks up
        double keep_p; // the probability to keep holding food
        double straight_p; // the probability to go straight
        double trust_p; // the probability to trust the latest pheromone info.
        unsigned int seed; // this ant’s seed used for rand_r()

        //To Report host0
        int temp_node_id; // to make the ant return to this node later
        int temp_daemon_id; // to make the ant return to this daemon later

        //IN decide direction
        bool activate_pheromon; // the flag to make the ant follow pheromone

62
```c
int temp_max;         // to store pheromone considered maximal temporarily
int temp_direction;   // to store the next moving direction temporarily

// Poisson Normal
double randVar;       // the latest random number obtained from rand_r()

void init();
void set_gene();
void synchronize();
void init_at_trail();
void work_at_trail();
void feel_pheromone(int tempMax, int tempDir, int dir);
void decide_direction();
void go_next_trail();

// Utility
int fact(int i);      // computes the factorial of i
int normal(double micro, double randVar);
int poisson(double lambda, double randVar);

// Utility Function
int Ant::fact(int i)   // computes the factorial of i
{
    int f;

    for (f = 1; i > 0; i--)
        f *= i;
    return(f);
}

int Ant::normal(double micro, double randVar) // compute the normal distribution
{
    const float e = 2.718281828459;
    const float pi = 3.141592;
    double n = 0.0;
    int i;

    for ( i = 0; i<=10; i++ ) {
        n += 1.0/sqrt( 2.0*pi ) * pow( e, -pow( i - micro, 2.0 )/2.0 );
        if ( n >= randVar )
            break;
    }
    cout << "normal=" << i << endl;
    return i;
}

int Ant::poisson(double lambda, double randVar) // compute the poisson distribution
{
    return i;
}
```

63
const double e = 2.718281828459;
double p = 0.0;
int i;

for (i = 0; i <= 10; i++) {
p += (pow(e, -beta) * (pow(beta, i) / fact(i)));
if (p >= randVar)
    break;
} //cout << "poisson" << i << endl;
return i;

void Ant::set_gene() {    // decide the ant gene

    int temp_gene;

    gene = random() & 0x0000ffff;
temp_gene = gene & (0x1);

    switch (temp_gene) {    // decide the ave. of pheromone reaction level
        case 0:
            norm_u=3.0;
            break;
        case 1:
            norm_u=4.0;
            break;
    }

temp_gene = temp_gene & (0x3);

    switch (temp_gene) {    // decide the initial pheromone reaction threshold
        case 0:
            ph_react0=3;
            break;
        case 1:
            ph_react0=4;
            break;
        case 2:
            ph_react0=5;
            break;
        case 3:
            ph_react0=6;
            break;
    }

temp_gene = temp_gene & (0x1);
switch (temp_gene) {  // decide the ave. amount of food which ant picks up
  case 0:
    pois_l=1.0;
    break;
  case 1:
    pois_l=2.0;
    break;
}

  temp_gene=gene >> 4;
  temp_gene=temp_gene & (0x3);

switch (temp_gene) {  // decide the probability to keep holding food
  case 0:
    keep_p=0.8;
    break;
  case 1:
    keep_p=0.85;
    break;
  case 2:
    keep_p=0.9;
    break;
  case 3:
    keep_p=0.95;
    break;
}
  temp_gene=gene >> 6;
  temp_gene=temp_gene & (0x3);

switch (temp_gene) {  // decide the probability to go straight
  case 0:
    straight_p=0.8;
    break;
  case 1:
    straight_p=0.85;
    break;
  case 2:
    straight_p=0.9;
    break;
  case 3:
    straight_p=0.95;
    break;
}
  temp_gene=gene >> 8;
  temp_gene=temp_gene & (0x3);

switch (temp_gene) {  // decide the probability to trust the latest pheromone
  case 0:
trust_p=0.8;
break;
case 1:
  trust_p=0.85;
  break;
case 2:
  trust_p=0.9;
  break;
case 3:
  trust_p=0.95;
  break;
}

seed = ( unsigned int )gene; // used for rand_r( )

if( DEBUG )
{
  printf("norm_u:%f\n",norm_u);
  printf("ph_react0:%d\n",ph_react0);
  printf("pois_1:%f\n",pois_1);
  printf("keep:%f\n",keep_p);
  printf("straight:%f\n",straight_p);
}

void Ant::init() // initialize the thread variables
{
  food=0;    // no food held
  step=0;   // time is 0
  direction=-1; // no direction decided
  total_food=0; // no food carried
  ph_react=0; // the initial pheromone reaction level is 0
  //Set Gene
  set_gene( ); // set my genetic information

  //move to nest
  ToNodeID = (daemonobj.meshsize/2)*daemonobj.meshsize
               + (daemonobj.meshsize/2); 
  hop( ToNodeID 0 daemonobj.getDaemonID( ToNodeID ) ); // hop to the nest
}

void Ant::synchronize() // visit daemon 0 to inject gvt
{
  if ( daemonobj.sleep_ant == 
       ( ( daemonobj.ant_num > daemon.total( ) ) ?
         ( daemonobj.ant_num / daemon.total( ) - 1 ) : 0 )
        && daemonobj.gvt_kicker_flag == false ) {
    // I AM GVT Kicker
daemonobj.gvt_kicker_flag=true;

    //Address to return to
temp_node_id= node.id();
temp_daemon_id=daemon.id() ;

    hop( INIT @ 0 );

    if( daemonobj.is_reported == false ) { // No one has injected gvt yet.
        daemonobj.is_reported = true;
        create thread<Gvt>;
    }
    hop( temp_node_id @ temp_daemon_id ); // go back where I was
}

daemonobj.sleep_ant++; // sleep there until gvt wakes me up
sleep daemon;
}

void Ant::init_at_trail() // the first thing at a new node
{
    node<Trail>.ant_in( );
    ph_react += ( daemonobj.ant_num/10 );
}

void Ant::work_at_trail() // the main work at a new node
{
    // get food
    if( node<Trail>.is_food() 
        && (node<Trail>.is_nest() != 1 )){
        randVar = fmod( rand_r( &seed ), 10000.0 ) / 10000.0;
        food = node<Trail>.get_food( poisson(pois_l, randVar) );
    }

    // drop food at nest
    if( node<Trail>.is_nest() == 1 ){
        if( food > 0 ) {
            node<Trail>.drop_food(food);
            total_food += food;
            food = 0;
        }
    }

    // drop food on route to nest
    randVar = fmod( rand_r( &seed ), 10000.0 ) / 10000.0;
    if( ( randVar > keep_p ) && food >= 1 ){
        node<Trail>.drop_food( 1 );
        food = food - 1;
void Ant::feel_pheromon( int& tempMax, int& tempDir, int& dir )
{
    // detect maximal pheromone and its
    int link_ph; // direction
    if ( food == 0 )
        link_ph = node<Trail>.get_pheromon_to_nest( );
    else
        link_ph = node<Trail>.get_pheromon_to_food( );

    if ( link_ph > tempMax )
    {
        tempMax = link_ph;
        tempDir = dir;
    }
}

void Ant::decide_direction() // decide which of eight links to hop along
{
    if( step == 0 ) // at time 0
        direction=rand_r( &seed ) % 8; // decide the direction randomly

    temp_max=0;
    temp_direction=0;

    if ( activate_pheromon ) // must follow pheromone
        /* checking if neighbors exist remotely or do not exist */
        border = false;
    for ( i = 0; i <= 7; i += 2 )
    {
        if ( !( link.exists(i) ) ) // check if link i does not exist
            if ( i == 0 & node<Trail>.y == 0 )
            {
                cout << "node[0,0]!!!" << endl;
                continue;
            }
        //cout << "link" << i << " not exist" << endl;
        border = true;
        break;
    }
    if ( i > 7 )
        // check if link i goes to a remote node
        for ( i = 0; i <=7; i += 2 )
        {
            if ( link[i].dest_daemon_id() != daemon.id( ) )
            {
                break;
            }
        }
    i = ( i + 7 ) % 8;
}
/* searching pheromon */
if ( !border ) {            // I am not on the border
    j=i;                    // detect each neighbor's pheromone
    hopalong( i );
    feel_pheromon( temp_max, temp_direction, j );
    ++j %= 8;
    hopalong( i = ( i + 3 ) % 8 );
    feel_pheromon( temp_max, temp_direction, j );
    ++j %= 8;
    hopalong( i );
    feel_pheromon( temp_max, temp_direction, j );
    ++j %= 8;
    hopalong( i = ( i + 2 ) % 8 );
    feel_pheromon( temp_max, temp_direction, j );
} else {                    // I am on the border
    i = ( i + 3 ) % 8;      // investigate only valid neighbors'
    if ( link.exists(i) ) {  // pheromone
        border = false;
        j=i;
        hopalong( i );
        feel_pheromon( temp_max, temp_direction, j );
        ++j %= 8;
        hopalong( i = ( i + 2 ) % 8 );
        feel_pheromon( temp_max, temp_direction, j );
    } else {
        border = true;
        i = ( i + 2 ) % 8;
        j=i;
        hopalong( i );
        feel_pheromon( temp_max, temp_direction, j );
    }
}
++j %= 8;
hopalong( i = ( i + 2 ) % 8 );
feel_pheromon( temp_max, temp_direction, j );

if ( !border ) {
    ++j %= 8;
    hopalong( i );
    feel_pheromon( temp_max, temp_direction, j );
}
++j %= 8;
hopalong( i = ( i + 2 ) % 8 );
feel_pheromon( temp_max, temp_direction, j );
++j %= 8;
hopalong( ( i + 2 ) % 8 );
}

randVar = fmod( rand_r( &seed ), 10000.0 ) / 10000.0; // get a random number
if ( randVar > trust_p || temp_max < ph_react ) {
  /* I don’t trust or can’t feel pheromone */
  do {
    /* I doubt my direction */
    direction = rand_r( &seed ) % 8;
    randVar = fmod( rand_r( &seed ), 10000.0 ) / 10000.0;
  } while ( randVar > straight_p );
} else {
  /* I’ll follow the pheromone */
  direction = temp_direction;
}

/* I’m on edge */
if(node<Trail>.x==0)
  direction=4;
else if(node<Trail>.x==daemonobj.meshsize-1)
  direction=0;
else if(node<Trail>.y==0)
  direction=2;
else if(node<Trail>.y==daemonobj.meshsize-1)
  direction=6;
}

void Ant::go_next_trail()
{
  // outward
  if( food == 0 ){
    randVar=fmod(rand_r( &seed ),10000.0)/10000.0; // get a new random number
    node<Trail>.drop_pheromon_to_food( normal(norm_u, randVar) );
    node<Trail>.ant_out(); // unregister itself at the current node
    hopalong(direction); // move to a next new node
  }

  // homeward
  if( food > 0 ){
    randVar=fmod(rand_r( &seed ),10000.0)/10000.0; // get a new random number
    node<Trail>.drop_pheromon_to_nest(normal(norm_u,randVar));
    node<Trail>.ant_out(); // unregister itself at the current node
    hopalong(direction); // move to a next new node
  }
}

void Ant::main() // the main ant behavior
{
  init(); // initialize its thread variables
}
//STEP
for ( step = 0; step < LIFE_STEP; step++ ) {
    synchronize();  // synchronized with gvt
    init_at_trail();  // register itself at a new node
    work_at_trail();  // pick up and drop off food
    decide_direction();  // search for food
    go_next_trail();  // go to a next new node
}

//To stop Timer
synchronize();
node<Trail>.ant_in();

if( DEBUG ) {
    cout << "ant[" << selfMigThr.getLocal() << "] got 
    << total_food << " amount of food"
    << "(norm_u=" << norm_u
    << " ph.react0=" << ph.react0
    << " pois.l=" << pois.l
    << " keep.p=" << keep_p
    << " straight.p=" << straight_p
    << " trust.p=" << trust_p
    << " seed=" << seed << ")"
    << endl;
}
}

5.2.5 Gvt Thread

The Gvt thread is injected by one of Ant threads every tick of simulation time and performs inter-threads synchronization. It repeats visiting every INIT node, (i.e., every daemon) until all Ant threads fall asleep, and wakes them up when confirming that they have slept. This thread carries the following three thread variables with it to achieve such synchronization:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant_sum</td>
<td>the total number of ants over the system</td>
</tr>
<tr>
<td>sum</td>
<td>the variable used to sum up the number of ants</td>
</tr>
<tr>
<td>i</td>
<td>a for-loop control variable</td>
</tr>
</tbody>
</table>

The gvt thread consists of the following two sub-method:

- **check_around** repeats visiting every INIT node and summing up the number of sleeping ants (=sum) until the number reaches the total number of ants
(\textit{=ant\_sum}).

- \textit{wake\_around} visits every \textit{INIT} node to wake up all ants sleeping on that daemon.

The following list is a complete code of the \textit{check\_result} thread.

```cpp
thread Gvt
{
public:
    Gvt( int argc, const char** argv ) : sum( 0 ) {}
    void main();
private:
    int ant\_sum; // # of ants over the system
    int sum; // sum up # ants on each daemon
    int i; // a for-loop control variable

    void check\_around(); // check if all ants are sleeping
    void wake\_around(); // wake up all ants sleeping on each daemon
};

void Gvt::check\_around() // check if all ants are sleeping
{
    ant\_sum = daemonobj.\textit{ant\_num}\n; // get the total number of ants
    while( 1 ){
        sum = 0;
        for ( i = 0; i < daemonobj.\textit{daemonTotal}; ++i ){
            hop( \textit{INIT} @ i ); // visit each daemon
            sum += daemonobj.\textit{sleep\_ant}; // sum up the number of ants sleeping there
        }
        if( sum == ant\_sum ) // if all ants are sleeping, return to main()
            break;
    }
}

void Gvt::wake\_around() // wake up all ants sleeping on each daemon
{
    for ( i = 0; i < daemonobj.\textit{daemonTotal}; ++i ){
        hop( \textit{INIT} @ i ); // visit each daemon
        if( daemonobj.gvt == 0 ) // if virtual time is 0 start the timer
            daemonobj.timer.\textit{start}( )
        daemonobj.gvt++;
        // increment the virtual time
        daemonobj.\textit{live\_ant} = daemonobj.\textit{sleep\_ant}; // all sleeping ants will be woken up.
        if ( daemonobj.gvt \% 500 == 0 ) // show #ants at this daemon even 500 cycles
            cout << "gvt" << daemonobj.gvt
                << ", \textit{ants}=" << daemonobj.\textit{live\_ant} << endl;
    }
    daemonobj.\textit{sleep\_ant} = 0; // no more sleeping ants
    daemonobj.is\_reported = false; // synchronization variables are reset

72
daemonobj.gvt_kicker_flag = false;
    wakeupall daemon;         // finally wake up all ants!

}

void Gvt::main()
{
    if ( daemonobj.gvt == LIFE_STEP ) {  // LIFE_STEP = 4001 in this implementation
        daemonobj.timer.stop();         // stop the timer
        cout <<"performance" <<endl;    // print out the performance data
        cout << daemonobj.timer.getElapsed() << endl;
        cout << daemonobj.timer.getInterval() << endl;
    }
    check_around();                    // otherwise, check if all ants are sleeping
    wake_around();                     // then, wake them up
    if ( daemonobj.gvt % 500 == 0 )    // print out the virtual time every 500 cycles
        cout << daemonobj.gvt <<endl;
}

5.2.6 CheckResult Thread

The CheckResult thread visits each node and displays its node status. This thread carries the following three thread variables with it:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j</td>
<td>the farm coordinates of each Tm if node</td>
</tr>
<tr>
<td>ToNodeID</td>
<td>the node ID computed from given i,j coordinates</td>
</tr>
</tbody>
</table>

The CheckResult thread computes the node ID from each pair of i,j coordinates, hops to it, and displays its node variables: x, y, food, pheromon_to_food, pheromon_to_nest, and ant_num.

A complete code of the check_result thread is shown below:

    thread CheckResult
    {
    public:
        void main();
    private:
        int i,j;
        int ToNodeID;
        void checkMesh();
    };
void CheckResult::checkMesh()
{
    for ( i = 0; i < daemonobj.meshsize; i++ ) {
        for ( j = 0; j < daemonobj.meshsize; j++ ) {
            ToNodeID = daemonobj.meshsize * i + j;
            hop(ToNodeID @ daemonobj.getDaemonID(ToNodeID));
            printf("mesh[%d,%d]: food=%d pheromon(ToNest=%d ToFood=%d) ants=%d\n",
                node<Trail>.x,
                node<Trail>.y,
                node<Trail>.food,
                node<Trail>.pheromon_to_food,
                node<Trail>.pheromon_to_nest,
                node<Trail>.ant_num
            );
        }
    }
}

void CheckResult::main()
{
    cout << "-----------------------" << endl;
    checkMesh();
}

5.2.7 Compilation and Execution

The ant farm shown above was programmed in /home/m++/appl/m++/ant.mpp. To compile this program, type:

    % m++ ant.mpp

To run this program, run inject commands as follows:

    % inject hostname ./Mesh 1 meshsize divider
    % inject hostname ./InitEnvironment 1 #ants
    % inject hostname ./Ant #ants 1
    % inject hostname ./CheckResult 1

where hostname, meshsize, divider, and #ants have the following meaning:

<table>
<thead>
<tr>
<th>parameters</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostname</td>
<td>the IP name of a computing node where each thread is injected</td>
</tr>
<tr>
<td>meshsize</td>
<td>the size of ant farm (meshsize * meshsize)</td>
</tr>
<tr>
<td>divider</td>
<td>1: divide ant farm in strip, or 0: divide ant farm in lattice</td>
</tr>
<tr>
<td>#ants</td>
<td>the total number of ants to be injected</td>
</tr>
</tbody>
</table>
Every time you start a new thread injection, we recommend you should wait for messages indicating that the previous injection and the subsequent thread action has completed.

6 Trouble Shooting

There are five possible errors you may encounter: abnormal termination, fopen error, bind system call error, segmentation fault/bus error, and dynamic module loading error. The following explains each error and its trouble shooting.

6.1 Abnormal Termination

An unexpected trap/interrupt or a Unix kill command may cause an abnormal termination and keep the previous socket connection in use. When it happens accidentally, you must wait for a few minutes until the underlying operating system recognizes that the previous socket port is already freed, or should edit your profile to change the port number.

6.2 Fopen error

This error occurs when an M++ daemon cannot locate profile. This file must be located in your current working directory where the daemon is invoked.

6.3 Bind system call error

This system call error occurs when a specified port number has been already allocated to socket communication. Possible scenarios and their respective solutions are:

1. You have already started an M++ daemon on this machine. Terminate the currently running daemon and restart a new daemon.

2. Someone is using the same port number to start his/her daemon. Redefine another port number in your profile.

3. Your previous M++ execution has resulted in an abnormal termination. Follow the instruction specified in Section6.1, “Abnormal Termination”.

75
6.4 Segmentation fault/Bus error

If the inject command causes this error, the number of arguments actually given is incompatible to that of arguments coded in your thread program. You should issue this command with the correct number of arguments. If a segmentation fault occurs during computation, your M++ threads but not daemons may have caused this fault. You should carefully look through the header and source files of your thread programs.

6.5 Dynamic module loading error

If the system displays a message, "SharedLib: Cannot load the dynamic module", an M++ daemon has resulted in failure of dynamic linking. There are two possible reasons to induce this error:

1. You might have given a wrong file to the inject command. In this case, simply re-inject your threads with an executable file name.

2. The daemon was unable to find out a class specified in [e]create node and/or [e]create link statements. Assure that you have defined a class corresponding to ClassName in the following statements:

   • [e]create node<ClassName>( args_list ) with ( NodeID[@DaemonID] )
   • [e]create link<ClassName>( args_list ) with ( SrcID ) to ( NodeID[@DaemonID] ) with ( DestID );

7 Final Comments

Focusing on multi-agent applications, the M++ system provides application designer with several important features: agent-based programming, runtime construction of simulation space, and virtual-time management. They can thus concentrate on modeling their multi-agent applications.

The M++ system is being tested by several users in University of Washington - Bothell, University of Tsukuba, University of California - Irvine, and Ehime University. M++ applications currently available include: ant farm, sugar mountain, remote exploration, and codi 1bit. Through more applications development and testing, we are planning to provide a debug environment, to enhance the system robustness, and implement dynamic load balancing.

Finally, we would like to mention that M++ is the outcome of our research on multithreading environments supported by Grant-in-Aid for Scientific Research No.
11780187, Japanese Society for the Promotion of Science. Special thanks to Dr. Luis Miguel Campos of UC Irvine. For the initial M++ functional and performance test, we have used his meta-computing simulation that motivated us to design the sthread and zero-copy thread migration libraries. We would also like to express our appreciation to Dr. Shinichi Yamagiwa and Mr. Mitsuhiro Ohto for their implementation of the zero-copy thread migration library. At the end, we would like to thank Mr. Yuichiro Tanaka, currently working for Fujitsu Co., for his assistance in programming M++ applications.

References


