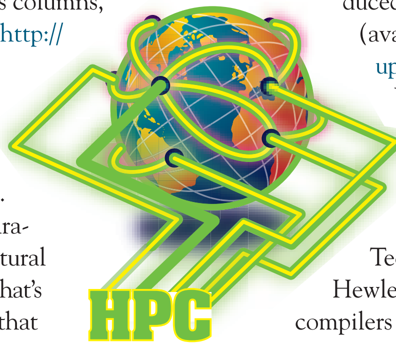
Unified Parallel C 101

By Forrest Hoffman

In addition to focusing on building, operating, and maintaining *Linux* clusters, this column has frequently explored the fundamentals of parallel programming. Message passing schemes running across distributed memory cluster nodes using both the *Message Passing Interface* (MPI) and *Parallel Virtual Machine* (PVM) have been presented. In addition, shared memory schemes within a multiprocessor node (using *OpenMP*) and hybrid combinations of MPI and OpenMP have been discussed. (To catch up on these previous columns, see the Parallel Programming "trail" at http://www.linux-mag.com/trails/parallel/.)

While most parallel codes must effectively utilize many distributed memory nodes, writing message passing code can be tedious and downright difficult. On the other hand, shared memory paradigms like OpenMP often seem more natural and are easier to code and maintain. What's needed is a parallel programming scheme that looks like a shared memory programming paradigm, but works across distributed memory hardware.

Enter *Unified Parallel C* (UPC).



Extending C

Unified Parallel C is an extension of the International Standards Organization (ISO) C 99 programming language. UPC unifies three previous modifications to the C language for supporting parallel programming: *AC*, *Split C*, and the *Parallel C Preprocessor*. Designed for high-performance computing on large-scale parallel machines, including Beowulf-style clusters, UPC provides a uniform programming model for both shared and distributed memory hardware. It uses a *single program, multiple data* (SPMD) model of computation, just like traditional message passing, in which the amount of parallelism is fixed at program startup time, usually with a single thread of execution for each processor.

In UPC, parallelism is expressed with an explicitly parallel execution model (SPMD with one or more *threads* executing a program in its entirety), a shared address space (through *shared* and *private* data type qualifiers), synchronization primitives (*locks*, *barriers*, and *memory fences*), a memory consistency model (*strict* and *relaxed* memory references), and memory management primitives. UPC is an attempt to combine the advantages of the shared memory programming paradigm with the control over data layout and performance of the message passing programming paradigm.

UPC is developed and supported by a consortium of universities, government laboratories, and computer vendors. Most notable among these are George Washington University, which hosts a community web site (http://upc.gwu.edu/), and Lawrence Berkeley National Laboratory (LBNL) which offers a UPC implementation and is working on a number of UPC-related projects (see http://upc.lbl.gov/). *Version 1.2* of a formal language specification for UPC was produced by the UPC Consortium in May 2005 (available online at http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf). In addition to LBNL's free UPC implementation, a free GCC implementation of UPC is available for *x86*, *x86_64*, SGI *IRIX*, and Cray *T3E* systems, and an MPI-based reference implementation is offered by Michigan Tech for Linux and *Tru64* operating systems. Hewlett Packard, Cray, and IBM also offer UPC compilers supporting their own hardware and operating systems.

The Basics of UPC Extensions to C

In UPC, every instance of execution is called a thread, as it is in OpenMP but unlike MPI, in which every instance of execution is called a process. UPC provides two important pre-defined identifiers that can be used in a program to determine the parallel layout during execution. `THREADS` is an integer that specifies the total number of threads that are executing the code. It has the same value on every thread.

LISTING ONE: *hello.upc*, a "Hello, World" program written in *Unified Parallel C*

```
#include <upc.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
   int i;

   for (i = 0; i < THREADS; ++i) {
     if (i == MYTHREAD)
       printf ("Hello World! I'm thread %d of %d\n",
         MYTHREAD, THREADS);
   }

   return 0;
}
```

LISTING TWO: *assn.upc* creates a shared array of integers

```
#include <upc.h>
#include <stdio.h>

shared int a[THREADS];

int main(int argc, char *argv[])
{
        int i;

        upc_forall (i = 0; i < THREADS; i++; i)
                a[i] = i;

        for (i = 0; i < THREADS; i++)
                printf ("%d: a[%d]=%d\n",
                  MYTHREAD, i, a[i]);

        return 0;
}
```

`MYTHREAD` is an integer that specifies the unique thread index, starting at zero, for each instance of execution.

Believe it or not, that's all you need to know to write your first UPC program, the equivalent of the "Hello, World! " program from previous MPI examples. *Listing One* contains the UPC version. Besides the *upc.h* header file and pre-defined identifiers `MYTHREAD` and `THREADS`, *Listing One* looks just like a standard C program. The program loops over the total number of threads and when the loop counter is equal to the thread number of the current instance of execution, "Hello, World! " is printed. All threads execute this entire loop and print a message only when each thread's local loop counter (`i`) is equal to `MYTHREAD`, the local thread number.

A key feature of UPC is that it uses a distributed shared memory model that provides for both private and shared memory spaces. Private memory is declared in normal C fashion, as was done for the variable `i` in *Listing One*. As mentioned in Ben Mayer's feature on next-generation parallel computing languages (in this issue), use of the shared memory space requires the `shared` qualifier on variable (or object) declarations as follows:

```
shared [block_size] type variable_name
```

For example, the line `shared int counter` declares an integer variable that resides in thread 0's memory but which is accessible by all threads. The line `shared int myarray [100]` is equivalent to `shared [1] int myarray[100]`. It defines an integer array that is shared across all threads with a *block size* of one. That means the first element of the array, `myarray[0]`, resides in thread 0's memory, `myarray[1]` resides in thread 1's memory, and so on, in round-robin fashion across all threads. The block size could be specified so that each thread holds one or more blocks of contiguous elements. A contiguous memory layout may provide better data locality for many codes; it may prevent frequent retrievals of data located in another thread's memory, significantly speeding up execution time.

For example, the line…

```
shared [N/THREADS] int parray[N]
```

… declares an integer array of `N` elements in blocks of size `N/THREADS`. The first block of `N/THREADS` elements resides with thread 0, the second block resides with thread 1, and so on. If `N` is not evenly divisible by `THREADS`, the `floor()` of `N/THREADS` is used, so an additional short block is created and assigned (in round-robin fashion) to thread 0.

A line of the form `shared [ ] int zarray[N]` declares an integer array of `N` elements with an infinite block size. This causes all elements of the array to be placed in the memory of thread 0. Shared pointers (private pointers to shared memory, shared pointers to shared memory, and even shared pointers to private memory) can be declared, but these constructs will be saved for a future column. In general, the association between memory and threads is called *affinity*.

As Ben Mayer suggests, knowledge of this affinity can be used to exploit data locality with constructs like the work sharing `upc_forall()` statement. This construct looks like a standard `for()` statement, but includes a fourth parameter that determines the thread to which the current iteration should be assigned. This parameter can be either an integer, which is translated to `(integer%THREADS)`, or an address, the owner of which is assigned the current iteration. Obviously this construct is useful only if loop iterations are independent of one another.

Listing Two shows a very simple UPC program that creates a shared integer array `a[]` with the number of elements being the number of threads in use. The program simply assigns to each element its element number and then each element of the array is printed. The program uses `upc_forall()` so that each thread assigns a value only to its own element of the `a[]` array; however, every thread subsequently prints every element of the shared array.

The `upc_forall()` statement in *Listing Two* assigns each iteration to the appropriate thread by taking its fourth parameter, `i`, and computing `i% THREADS`. The statements could have been written equivalently as…

```
upc_forall (i = 0; i < THREADS; i++; &a[i])
```

In that case, each loop iteration is automatically assigned to the thread on which `a[i]` resides.

This program looks straightforward enough, but it has a bug. Care to guess what it is?

Before answering that question, let's get a compiler (or two) and start running the code.

Getting a Compiler

A number of UPC compilers are available for Linux systems. The two most obvious candidates are GCC UPC (available at http://www.intrepid.com/upc/) and *Berkeley UPC* (available at http://upc.nersc.gov/). GCC UPC is essentially a patch to the GCC sources and is implemented as a C language dialect translator in the same fashion as the GNU *Objective C* compiler. GNU UPC is available for Intel *Itanium* (*ia64*) Linux, AMD (*amd64*) Linux, Intel (x86) Linux, SGI IRIX, and Cray T3E. It can be built by patching existing GCC sources and compiling, by downloading a fully patched suite of GCC sources and compiling, or by downloading and installing binaries. It may be easiest to simply build a custom version of GCC in a separate place to try out this compiler. This can be accomplished by following these steps as *root*:

FIGURE ONE: Testing the *Berkeley UPC* compiler

```
[upc]$ export PATH=/usr/local/berkeley_upc-2.2.1/bin:$PATH
[forrest@node01 upc]$ upcc --version
This is upcc (the Berkeley Unified Parallel C compiler), v. 2.2.1
   (getting remote translator settings...)

  UPC Runtime           | v. 2.2.1, built on Jan 13 2006 at 00:48:33
  UPC-to-C translator   | release 2.2.0, built on Dec  2 2005 at 07:26:48
  Translator location   | http://upc-translator.lbl.gov/upcc-2.2.cgi
  networks supported    | udp smp mpi
  default network       | mpi
        …
  linker flags          | -DGASNET_NDEBUG -O3 -finline-limit=10000 -Winline
                         | -L/usr/local/berkeley_upc-2.2.1/lib -lupcr-mpi-seq
                         | -lumalloc -L/usr/local/berkeley_upc-2.2.1/lib
                         | -lgasnet-mpi-seq -lammpi
                         | -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 -lgcc -lm

[upc]$ upcc —network=udp -o hello hello.upc
[upc]$ export UPC_NODES="node13 node14 node15 node16 \
  node17 node18 node19 node20 node21 node22"
[upc]$ export UPC_SSH=rsh
[upc]$ upcrun -n 4 ./hello
UPCR: UPC thread 0 of 4 on node13 (process 0 of 4, pid=4687)
UPCR: UPC thread 1 of 4 on node14 (process 1 of 4, pid=3835)
UPCR: UPC thread 2 of 4 on node15 (process 2 of 4, pid=3487)
UPCR: UPC thread 3 of 4 on node16 (process 3 of 4, pid=3687)
Hello World! I'm thread 1 of 4
Hello World! I'm thread 3 of 4
Hello World! I'm thread 0 of 4
Hello World! I'm thread 2 of 4
```

```
[root]# cd /usr/local/src
[src]# wget ftp://ftp.intrepid.com/pub/upc/
   rls/upc-3.4.4.1/upc-3.4.4.1.src.tar.gz
[src]# tar xvzf upc-3.4.4.1.src.tar.gz
[src]# cd upc-3.4.4.1
[upc-3.4.4.1]# mkdir -p /usr/local/upc-3.4.4.1
[upc-3.4.4.1]# ./configure
  --prefix=/usr/local/upc-3.4.4.1
[upc-3.4.4.1]# make; make install
```

To try UPC, compile and run the "Hello, World! " program. Since the steps above didn't install the compiler in the normal location for system binaries, you'll need to add the location of the compiler binaries to your path before compiling and running the program.

```
[upc]$ export PATH=/usr/local/
  upc-3.4.4.1/bin:$PATH
[upc]$ upc -fupc-threads-4 hello.upc -o hello
[upc]$ ./hello
Hello World! I'm thread 0 of 4
Hello World! I'm thread 1 of 4
Hello World! I'm thread 2 of 4
Hello World! I'm thread 3 of 4
```

It appears to work. Four threads, specified at compile time, started and printed `Hello World!` But all four threads ran on the same node, and this node has only two processors. For a real parallel model, the goal would be to have one thread per processor, and have the model running on multiple nodes. Unfortunately, there is no network layer in GCC UPC to support running across distributed memory nodes. For an SGI Altix or a Cray T3E, each of which appears as one large symmetric multiprocessor system, GCC UPC would work fine, but for Beowulf-style clusters another solution is needed.

Fortunately, Berkeley UPC is built on top of their *GASNet* portable networking library. It supports not only an SMP configuration, but also works on top of MPI or over *Ethernet UDP*, *Myrinet GM*, *Quadrics ELAN 3/4*, *Mellanox Infiniband VAPI*, *IBM LAPI*, *Dolphin SCI*, and *SHMEM* (on SGI *Altix* and Cray *X1* systems). Native support for this wide array of high-bandwidth, low-latency interconnects means that UPC could be used for serious high-performance computing applications. The runtime system has been tested on Linux, *FreeBSD*, *NetBSD*, *Tru64*, *AIX*, *IRIX*, *HPUX*, *Solaris*, *Microsoft Windows/Cygwin*, *Mac OS X*, *Cray Unicos*, and *NEC SuperUX*.

The Berkeley compiler is really a runtime/front-end program that communicates with a UPC-to-C translator. Interestingly, LBNL allows public access to their translator via HTTP, since the translator can be built on a narrower range of systems. By default, the front-end program simply uses the LBNL translator over the network. The front-end can be downloaded and built by following these steps as *root*:

```
[root]# cd /usr/local/src
[src]# wget http://upc.lbl.gov/download/
  release/berkeley_upc-2.2.1.tar.gz
[src]# tar xvzf berkeley_upc-2.2.1.tar.gz
[src]# cd berkeley_upc-2.2.1
[berkeley_upc-2.2.1]# mkdir -p
  /usr/local/berkeley_upc-2.2.1
[berkeley_upc-2.2.1]# ./configure
  --prefix=/usr/local/berkeley_upc-2.2.1
[berkeley_upc-2.2.1]# make; make install
```

These steps build in support only for SMP, UDP, and MPI (if it is available on the system). Additional flags must be provided to *configure* to compile in support for the other network hardware listed above.

Again, because the software was installed in an isolated place, you and your users will need to add this location to their path. Running `upcc --version` provides detailed information about the runtime system. The steps shown in *Figure One* can be used to test the compiler using the "Hello, World! " program.

In *Figure One*, UDP is selected as the network to use at compile time. Apparently UDP has better performance than MPI. A list of nodes was provided in the `UPC_NODES` environment variable, and the default program for contacting the other nodes was changed to *rsh* by setting the `UPC_SSH` environment variable.

With Berkeley UPC, the compiled code must be executed using the *upcrun* command. The `-n` flag is used to set the number of threads desired (unless that number is set at compile time). Four threads were requested, and those were all spawned on separate nodes.

Now the code in *Listing Two* can be tested. Using the GCC UPC compiler with four threads, compiling and running the *assn* program yields:

```
[gcc_upc]$ upc -fupc-threads-4 -o
  assn assn.upc
[gcc_upc]$ ./assn | sort
0: a[0]=0
0: a[1]=1
0: a[2]=0
0: a[3]=0
1: a[0]=0
1: a[1]=1
1: a[2]=0
1: a[3]=0
2: a[0]=0
2: a[1]=1
2: a[2]=2
2: a[3]=0
3: a[0]=0
3: a[1]=1
3: a[2]=2
3: a[3]=3
```

Here, the problem is evident. Not all of the values of `a[2]` and `a[3]` are correct, but this is one big shared array. The problem is that some threads print the values before the values are set by other threads. Something is needed to make each thread wait until all the threads have finished setting values in the shared array.

This can be accomplished by inserting a `upc_barrier` between the `a[i]=i;` line and the `for(i=0; i< THREADS; i++)` line in *Listing Two*. When the program is recompiled and run, the following output is generated:

```
0: a[0]=0
0: a[1]=1
0: a[2]=2
0: a[3]=3
1: a[0]=0
1: a[1]=1
1: a[2]=2
1: a[3]=3
2: a[0]=0
2: a[1]=1
2: a[2]=2
2: a[3]=3
3: a[0]=0
3: a[1]=1
3: a[2]=2
3: a[3]=3
```

Try It Yourself!

That should be enough information to get you started using UPC. Give it a try and see if it makes programming your parallel codes easier. You can also read the book *UPC: Distributed Shared Memory Programming* (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471220485, descCd- description.html) for in-depth specifications, examples, and lots of working code.

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov.