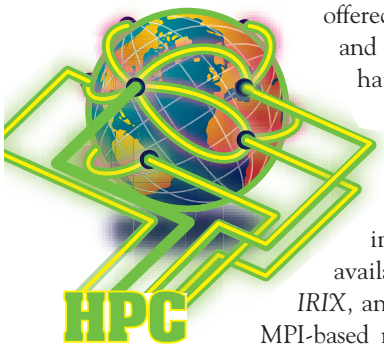
Unified Parallel C, Part Two

By Forrest Hoffman

The March 2006 "Extreme Linux" column — and a March feature story by Ben Mayer — introduced the *Unified Parallel C* (UPC) language. Languages like UPC (and *Co-Array Fortran* and others) are designed to make parallel programming easier and make the resulting code more maintainable by making parallelism more implicit (like shared-memory paradigms) and less explicit and cumbersome (like message passing schemes). Using UPC, programmers can often write code more quickly and with fewer errors, while still maintaining control over data layout.

UPC, an extension of the International Standards Organization (ISO) *C 99* programming language, 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. UPC is developed and supported by a consortium of universities, government laboratories, and computer vendors. In addition to the UPC compilers offered by Hewlett Packard, Cray, and IBM (supporting their own hardware and operating systems), a few free implementations are also available.



A free GCC-based 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 the *Linux* and *Tru64* operating systems. However, the most popular implementation for Beowulf-style clusters appears to be the one offered by the Lawrence Berkeley National Laboratory (LBNL). *Berkeley UPC* is built on top of their *GASNet* portable networking library, so it supports not only a symmetric multiprocessor (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).

Berkeley UPC's native support for a wide array of high bandwidth, low latency interconnects makes it ideal for serious high performance computing applications on Beowulf-style Linux clusters. 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 only on a small range of systems. However, the runtime system runs on Linux, *FreeBSD*, *NetBSD*, Tru64, *AIX*, *IRIX*,

LISTING ONE: *mvmult1.upc*, an example matrix-vector multiplication

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

shared double a[THREADS][THREADS];
shared double b[THREADS], c[THREADS];

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

  for (i = 0; i < THREADS; i++)
    upc_forall (j = 0; j < THREADS; j++; j)
      a[i][j] = (double)(i*THREADS+j+1);
  upc_forall (j = 0; j < THREADS; j++; &b[j])
    b[j] = (double)j+1;

  upc_barrier;

  for (i = 0; i < THREADS; i++) {
    c[i] = 0.;
    /* WARNING: THIS IS A PROBLEM! */
    upc_forall (j = 0; j < THREADS; j++; j)
      c[i] += a[i][j] * b[j];
  }

  upc_barrier;

  if (MYTHREAD == 0) {
    printf("+-     ");
    for (i = 1; i < THREADS - 1; i++) printf("      ");
    printf("     -+   +-   -+    +-     -+\n");
    for (i = 0; i < THREADS; i++) {
      printf("|");
      for (j = 0; j < THREADS; j++) {
        printf(" %4.1lf", a[i][j]);
      }
      printf("|  |%4.1f| ", b[i]);
      if (i == (int)(THREADS/2)) printf("=");
        else printf(" ");
      printf(" |%6.1f|\n", c[i]);
    }
    printf("+-     ");
    for (i = 1; i < THREADS - 1; i++) printf("      ");
    printf("     -+   +-   -+    +-     -+\n");
  }

  return 0;
}
```

HPUX, *Solaris*, *Windows/Cygwin*, *Mac OS X*, *Cray Unicos*, and *NEC SuperUX*.

You can find instructions for downloading, building, and installing both GCC UPC and Berkeley UPC in the March stories. You can also find examples of using `upc_forall()` and `upc_barrier()` in those articles. This month, let's apply UPC to a more interesting problem that further illustrates key features of the language.

Data Distribution with UPC

Recall that UPC uses a distributed shared memory model that provides for both private and shared memory spaces, and that portions of the globally shared address space have a static affinity with a thread. Knowledge of this affinity can be used to exploit the efficiency of data locality. Consider, for an example (and to review last month's discussion), a piece of code that performs matrix-vector multiplication, shown in *Listing One*.

This program initializes an array (`a`) and a vector (`b`), solves `a` times `b`= `c`, and prints the results. The code starts by including *upc_relaxed.h*, which specifies the relaxed memory consistency mode.

This mode is in contrast to strict mode for which *upc_strict.h* should be included. In strict mode, shared data are synchronized prior to access by another thread. Strict mode also prevents the compiler from rearranging operations utilizing shared data, and it can result in significant overhead. As a result, relaxed mode is preferred, but it requires that the programmer ensure memory consistency through the use of fences, barriers, and locks.

Next, the matrix (`a`) and both vectors (`b` and `c`) are declared as shared, double precision data objects. The `THREADS` keyword is the number of threads running the code, and, in this case, it must be specified at compile time.

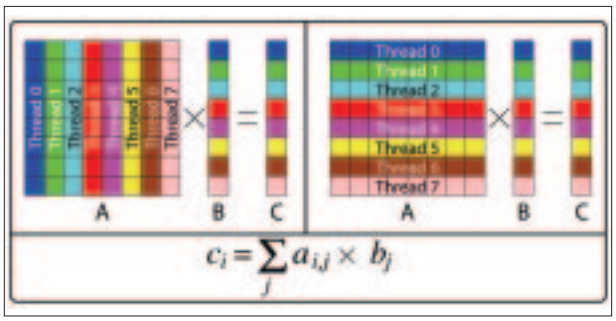


FIGURE ONE: Two strategies to distribute work to threads

FIGURE TWO: Differing results from execution to execution aren't very useful

```
[gcc_upc]$ upc -fupc-threads-8 -o mvmult1 mvmult1.upc
[gcc_upc]$ ./mvmult1
+-
|  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0|  | 1.0|   |  49.0|
|  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0|  | 2.0|   | 105.0|
| 17.0 18.0 19.0 20.0 21.0 22.0 23.0 24.0|  | 3.0|   | 161.0|
| 25.0 26.0 27.0 28.0 29.0 30.0 31.0 32.0|  | 4.0|   | 217.0|
| 33.0 34.0 35.0 36.0 37.0 38.0 39.0 40.0|  | 5.0| = | 273.0|
| 41.0 42.0 43.0 44.0 45.0 46.0 47.0 48.0|  | 6.0|   | 329.0|
| 49.0 50.0 51.0 52.0 53.0 54.0 55.0 56.0|  | 7.0|   | 385.0|
| 57.0 58.0 59.0 60.0 61.0 62.0 63.0 64.0|  | 8.0|   | 441.0|
+-
[gcc_upc]$ ./mvmult1
+-
|  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0|  | 1.0|   |   9.0|
|  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0|  | 2.0|   |  33.0|
| 17.0 18.0 19.0 20.0 21.0 22.0 23.0 24.0|  | 3.0|   |  57.0|
| 25.0 26.0 27.0 28.0 29.0 30.0 31.0 32.0|  | 4.0|   |  81.0|
| 33.0 34.0 35.0 36.0 37.0 38.0 39.0 40.0|  | 5.0| = | 105.0|
| 41.0 42.0 43.0 44.0 45.0 46.0 47.0 48.0|  | 6.0|   | 129.0|
| 49.0 50.0 51.0 52.0 53.0 54.0 55.0 56.0|  | 7.0|   | 153.0|
| 57.0 58.0 59.0 60.0 61.0 62.0 63.0 64.0|  | 8.0|   | 177.0|
+-
```

Inside `main()`, `upc_forall()` is used to initialize `a` and `b`. Since no block size was specified for these shared data objects, the default block size of 1 is used and each element of the matrix and vector is assigned in round robin fashion to threads, as shown in the left half of *Figure One*.

Exploiting Data Affinity

Knowing this affinity, initialization is optimized by having each thread initialize only the portion of the shared data objects local to it. For the matrix `a`, initialization for a column is assigned to each thread in turn, while for `b`, initialization of each row element is assigned to each thread in turn. After these two `upc_forall()` loops, `upc_barrier()` is called to provide the needed synchronization point. This ensures that no other work is done until all the threads are finished initializing `a` and `b`.

After the barrier, the matrix-vector multiplication is performed in parallel, according to the equation at the bottom of *Figure One*. Again, the `upc_forall()` loop is written so that, with the default block size of 1, the thread to which elements of `a` and `b` have affinity perform the multiplication. Next, another `upc_barrier()` call is made before thread 0 prints out the entire matrix and both vectors. Then the program exits by returning 0.

However, there's an evident problem when the program is

compiled with GCC UPC and run twice, as shown in *Figure Two*. Programs that generate different results when run twice aren't very useful. In both cases, the `a` matrix and `b` vector are identical, but the solution (the `c` vector) is different. The problem with *mvmult1.upc* is that the `j` loop in the matrix-vector multiplication is not parallel, since each thread updates `c[i]` simultaneously.

The matrix-vector multiplication can, however, be parallelized by parallelizing the `i` loop. In that case, every element of `c` is computed by a single thread. In fact, each thread computes the `c[i]` which has affinity with that thread (i.e., the local `c[i]`). Instead of having elements of `b` local, elements of `c` are local.

With the `i` loop parallel, each thread computes an element of `c` using a row of `a` instead of a column, which makes more sense. Therefore, each thread must obtain all but one element of its row from the other threads unless the `a` matrix is distributed differently. Fortunately, this is easy to do in UPC. Simply change the declaration of the `a` matrix to give it a block size of `THREADS` as follows:

```
shared [THREADS] double
a[THREADS][THREADS];
```

With that small change, the matrix is distributed among the threads as shown in the right half of *Figure One*. In the distribution, each thread needs to obtain only the (`THREADS`- 1) elements of `b` from other threads to compute its element of `c`, since all the required elements of `a` are local.

FIGURE THREE: The results of running *Listing Two*

```
[gcc_upc]$ upc -fupc-threads-8 -o mvmult2 mvmult2.upc
[gcc_upc]$ ./mvmult2
+-
|  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0|  | 1.0|   | 204.0|
|  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0|  | 2.0|   | 492.0|
| 17.0 18.0 19.0 20.0 21.0 22.0 23.0 24.0|  | 3.0|   | 780.0|
| 25.0 26.0 27.0 28.0 29.0 30.0 31.0 32.0|  | 4.0|   |1068.0|
| 33.0 34.0 35.0 36.0 37.0 38.0 39.0 40.0|  | 5.0| = |1356.0|
| 41.0 42.0 43.0 44.0 45.0 46.0 47.0 48.0|  | 6.0|   |1644.0|
| 49.0 50.0 51.0 52.0 53.0 54.0 55.0 56.0|  | 7.0|   |1932.0|
| 57.0 58.0 59.0 60.0 61.0 62.0 63.0 64.0|  | 8.0|   |2220.0|
+-
```

FIGURE FOUR: Running *Listing Two* on many processors

```
[bupc]$ upcc --network=udp -T=8 -o mvmult2 mvmult2.upc
[bupc]$ export UPC_NODES="node13 node14 node15 node16 \
  node17 node18 node19 node20 node21 node22"
[bupc]$ upcrun ./mvmult2
UPCR: UPC thread 0 of 8 on node13 (process 0 of 8, pid=3765)
UPCR: UPC thread 2 of 8 on node15 (process 2 of 8, pid=2643)
UPCR: UPC thread 6 of 8 on node19 (process 6 of 8, pid=32290)
UPCR: UPC thread 4 of 8 on node17 (process 4 of 8, pid=5362)
UPCR: UPC thread 1 of 8 on node14 (process 1 of 8, pid=3180)
UPCR: UPC thread 7 of 8 on node20 (process 7 of 8, pid=32005)
UPCR: UPC thread 3 of 8 on node16 (process 3 of 8, pid=2432)
UPCR: UPC thread 5 of 8 on node18 (process 5 of 8, pid=1818)
+-
|  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0|  | 1.0|   | 204.0|
|  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0|  | 2.0|   | 492.0|
| 17.0 18.0 19.0 20.0 21.0 22.0 23.0 24.0|  | 3.0|   | 780.0|
| 25.0 26.0 27.0 28.0 29.0 30.0 31.0 32.0|  | 4.0|   |1068.0|
| 33.0 34.0 35.0 36.0 37.0 38.0 39.0 40.0|  | 5.0| = |1356.0|
| 41.0 42.0 43.0 44.0 45.0 46.0 47.0 48.0|  | 6.0|   |1644.0|
| 49.0 50.0 51.0 52.0 53.0 54.0 55.0 56.0|  | 7.0|   |1932.0|
| 57.0 58.0 59.0 60.0 61.0 62.0 63.0 64.0|  | 8.0|   |2220.0|
+-
```

With this new scheme, the `a` matrix would optimally be initialized with a parallel loop over `i` instead of over `j` as well. *Listing Two* contains the corrected code with the appropriate block size and parallel loops. (The serial code that prints out every element of the matrix and the vectors isn't repeated in *Listing Two;* it's the same as in *Listing One*.)

As shown in *Figure Three*, the new code generates the correct output when compiled with GCC UPC and re-run. In fact, this code generates correct answers even if the block size of `a` were not specified, since the parallelism is done correctly. However, more communication would be required, making the code less efficient.

In all of these tests, the programs were compiled for 8 threads, but run on a machine that has only two processors. For real parallel codes, either a large SMP machine is needed or a different version of UPC is required to distribute the threads among other Linux cluster nodes.

Berkeley UPC does this very thing using its GASNet network layer. *Figure Four* shows one way to compile and run the code using Berkeley UPC. Each of the eight threads was run on a different node, and the generated results are correct.

The Berkeley UPC compiler front-end allows the programmer to choose a desired network using the `--network` parameter. The *User Datagram Protocol* (UDP) is the most efficient on Linux clusters not employing a high bandwidth, low latency interconnect. The number of threads is set using

`-T`, although, to quote the *upcc* man page, "The disgusting syntax `-F (upc-) threads-`*`NUM`* is also accepted, for compatibility with other UPC compilers."

You Can Try This at Home!

Getting the data distributed properly for optimal efficiency in computation is an important first step in writing parallel code in any language. UPC's method for distributing data is pretty easy, and it allows you to avoid writing explicit message passing code. With your favorite algorithm in hand, try out UPC for yourself. It might be just what you need to produce an efficient parallel program without lots of MPI calls.

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov. You can download the code shown here from http://www.linux-mag.com/download/ 2006-05/extreme.tar.gz.

LISTING TWO: *mvmult2.upc,* an improved matrix-vector multiplication code

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

shared [THREADS] double a[THREADS][THREADS];
shared double b[THREADS], c[THREADS];

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

  upc_forall (i = 0; i < THREADS; i++; i)
    for (j = 0; j < THREADS; j++)
      a[i][j] = (double)(i*THREADS+j+1);
  upc_forall (j = 0; j < THREADS; j++; &b[j])
    b[j] = (double)j+1;

  upc_barrier;

  upc_forall (i = 0; i < THREADS; i++; i) {
    c[i] = 0.;
    for (j = 0; j < THREADS; j++)
      c[i] += a[i][j] * b[j];
  }

  upc_barrier;

  /* Insert here the block of code from
   * mvmult1.upc to print out the arrays */

  return 0;
}
```