Unified Parallel C, Part Three

By Forrest Hoffman

This is the third column introducing the basics of *Unified Parallel C* (UPC). UPC, *Co-Array Fortran*, and other new productivity-oriented programming languages are designed to simplify parallel programming and code maintenance. For some types of algorithms, UPC allows programmers to code more quickly and in a style that is considerably less cumbersome than traditional message passing.

As described previously, UPC is an extension of the International Standards Organization (ISO) C 99 programming language. UPC uses a Single Program, Multiple Data (SPMD) model of computation, just like message passing. UPC is developed and supported by a consortium of universities, government laboratories, and computer vendors. Commercial UPC compilers are available from Hewlett Packard, Cray, and IBM; however, a number of free implementations are also becoming available.

Part One of this series provided instructions for building and installing two of the free UPC implementations, one based on GCC (and available for *x86*, *x86_64*, *SGI IRIX*, and *Cray T3E* systems), and one available from the Lawrence Berkeley National Laboratory (LBNL), which builds on LBNL's *GASNet* portable networking library. The GCC implementation is good for codes running solely on SMP (symmetric multi-processor) machines, while the Berkeley implementation works well on distributed memory parallel systems and supports a wide variety of network interconnects. The Berkeley compiler is really a runtime/front-end program that communicates with a UPC-to-C translator. By

LISTING ONE: The matrix-matrix multiply code, *mmmult.upc*

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

#define N         4
#define P         3
#define M         6

/* N and M must be multiples of THREADS */
shared [N/THREADS*P] double a[N][P];
 /* decomposed row-wise */
shared [M/THREADS]    double b[P][M];
/* decomposed column-wise */
shared [N/THREADS*M] double c[N][M];
/* decomposed row-wise */
double b_local[P][M];

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

    /* Initialization of a and b */
    upc_forall (i = 0; i < N; i++; &a[i][0])
      for (j = 0; j < P; j++)
        a[i][j] = (double)(i*P+j+1);
      upc_forall (j = 0; j < M; j++; &b[0][j])
    for (i = 0; i < P; i++)
        b[i][j] = (double)((P*M+1)-(i*M+j+1));

    upc_barrier;

/* Cheaper to pre-fetch elements of b into local
copy instead of implicitly fetching them every time
they are needed */
  for (i = 0; i < P; i++)
    for (j = 0; j < THREADS; j++)
      upc_memget(&b_local[i][j*(M/THREADS)],
        &b[i][j*(M/THREADS)],
        (M/THREADS)*sizeof(double));

  upc_forall (i = 0; i < N; i++; &c[i][0]) {
    for (j = 0; j < M; j++) {
      c[i][j] = 0.;
      for (k = 0; k < P; k++)
        c[i][j] += a[i][k] * b_local[k][j];
    }
  }

  upc_barrier;

  if (MYTHREAD == 0) {
    printf("    +-      ");
    for (i = 1; i < M - 1; i++) printf("       ");
    printf("     -+\n");
    for (i = 0; i < N; i++) {
      if (i == (int)(N/2) - 1) printf("c = ");
      else printf("    ");
      printf("|");
      for (j = 0; j < M; j++)
        printf(" %5.1lf", c[i][j]);
      printf("|\n");
    }
    printf("    +-      ");
    for (i = 1; i < M - 1; i++) printf("       ");
    printf("     -+\n");
  }

  return 0;
}
```

default, it accesses the translator at LBNL via *HTTP*.

The `upc_forall()` and `upc_barrier()` calls were described and used in example programs in Part One (available online at http://www.linux-mag.com/2006-03/extreme_01.html). May's column (available online after June 15, 2006, at http://www.linux-mag.com/2006-03/extreme_01.html) discussed data distribution and thread affinity and applied these concepts in a parallel code that performed a simple matrix-vector multiplication.

This month's column includes a more-useful matrix-matrix multiplication example and describes some additional features of the UPC language.

Matrix-Matrix Multiplication

UPC employs a distributed shared memory model that has both private and shared memory spaces, and portions of the shared memory have a static affinity with a thread. Previous example programs presented here have attempted to demonstrate the best ways to exploit this data locality by appropriately adjusting blocking of shared objects. In the matrix-vector multiply program, it was a simple matter to change the blocking on the matrix so that it decomposed row-wise instead of column-wise.

Similarly, in the matrix-matrix multiply code shown in *Listing One*, which performs *AxB*=C, the double-precision `N x P` matrix `A` is decomposed row-wise, while the double-precision `P x M` matrix `B` is decomposed column-wise. This is accomplished with the declarations of `a` and `b` as `shared double`, with the appropriate blocking factors `[N/THREADS *P]` and `[M/THREADS]`, respectively.

With these declarations, `N/THREADS` consecutive rows of `a` are placed on each thread, while `M/THREADS` consecutive columns of `b` are placed on each thread. This assumes, of course, that `N` and `M` are multiples of `THREADS`. For simplicity, assume that two threads are used, and that `N`, `P`, and `M` are statically defined at the top of *Listing One*.

Considering that the first `N/THREADS` rows of `a` and the first `M/THREADS` columns of `b` have affinity with the first thread, one would expect the best performance to be achieved with a row-wise decomposition for the double-precision `N x M` matrix `C` as shown in the declaration of `c` in *Listing One*. However, every row of `a` must be multiplied by every column of `b`, so some communication of elements of `b` is required to complete the calculation. For this reason, a local (non-shared) matrix `b_local` is declared.

$$A \times B = C$$

$$c_{i,j} = \sum_{k=1}^{P} a_{i,k} \times b_{k,j}$$

FIGURE TWO: The formula for matrix multiplication

FIGURE ONE: Memory copy and memory set routines in *Unified Parallel C*

- `upc_memcpy(shared void*restrict dst, shared const void*restrict src,size_t n)` copies `n` bytes from a shared object having affinity with one thread to a shared object having affinity with the same or another thread.
- `upc_memget(void*restrict dst, shared const void*restrict src, size_t n)` copies `n` bytes from a shared object with affinity to any single thread to an object on the calling thread.
- `upc_memput(shared void*restrict dst, const void*restrict src, size_t n)` copies `n` bytes from an object on the calling thread to a shared object with affinity to any single thread.
- `upc_memset(shared void*dst, int c, size_t n)` copies the value of `c`, converted to an `unsigned char`, to a shared object with affinity to any single thread. The number of bytes set is `n`.

UPC and other new productivity-oriented programming languages are designed to simplify parallel computing

Inside `main()`, the `a` and `b` matrices are initialized in parallel, based on their specified decompositions, using `upc_forall` loops. Rows of `a` and columns of `b` are set to some value on the thread to which they have affinity. Next, a `upc_barrier` call ensures that `a` and `b` are completely initialized before the program continues. After the barrier, block copies of the shared `b` matrix — accomplished with the `upc_memget()` call — explicitly copy elements of the `b` matrix to the `b_local` matrix, which is private on each thread.

While it's not necessary for threads to have a local copy of the `b` matrix, a great deal more communication occurs in the computation of the `c` matrix (below) if each thread must repeatedly retrieve elements of `b` from other threads. It's more efficient to retrieve these elements once, and use a local copy of the `b` matrix in the computation of `c`. The `upc_memget()` routine, which looks very much like a one-sided message passing communication call, provides a method to explicitly

copy shared data to private memory. Memory copy and set routines in UPC are listed in *Figure One*.

After each thread has constructed its own copy of `b_local`, elements of the resulting `c` matrix are computed in parallel using the now-familiar `upc_forall()` loop construct following the equation in *Figure Two*. The work is split up according to the affinity of each row of `c`. Since the affinity of `a` matches that of `c` (and since a local copy of `b` is used), this computation fully exploits data locality.

Another `upc_barrier` call is made to ensure that all of the elements of `c` are computed on all threads before the first thread (`MYTHREAD==0`) prints out the contents of the `c` matrix. Finally, all threads return zero and the program ends.

The Correct Placement of Barriers

Two of the pitfalls of the simplicity of UPC are forgetting to include and misplacing barrier calls. Since it's very easy to implicitly retrieve data objects from other threads in UPC, one must be careful to guarantee that these data objects are up-to-date. The matrix-matrix multiplication program offers a simple example of the potential dangers.

In the initial version of this code, the first `upc_barrier` call was accidentally placed after the loop that pre-fetches elements of `b` into the private `b_local` matrix using `upc_memget` instead of before it. When the program was compiled and run using GCC UPC with two threads, as shown in *Figure Three*, it generated the `c` matrix, but six of the elements of this matrix were zero. By printing the values of all the other matrices, it was easy to determine the problem.

The first thread retrieved elements of `b` from the second thread before they were initialized by the program. As a result, the `b_local` matrix on the first thread contained all zeroes in the fourth, fifth, and sixth columns, and these zeroes were used in the calculation of the fourth, fifth, and sixth columns of the first two rows in `c`, as shown in *Figure Three*.

Moving the `upc_barrier` call up to just after the initialization of `b` but before the loop with `upc_memget()` causes the program to wait for all threads to complete the initialization of `a` and `b` before continuing. Once all threads reach the barrier, it's safe to fetch elements of `b` from any thread. *Figure Four* shows the correct output from the program compiled and run using two threads with GCC UPC and with Berkeley UPC.

Pointers and Dynamic Memory Allocation in UPC

UPC provides for both private and shared pointers, which may point to either private or shared objects. The declaration `shared int *p;` defines a pointer to an integer residing in the shared memory space. Pointer arithmetic is possible with shared pointers, but it may be slow, since it follows the shared object throughout shared memory. In many cases, better performance may be achieved by using a private pointer into shared memory that has affinity with a given thread.

Global memory can be allocated and freed explicitly with UPC routines as follows:

```
shared void*upc_global_alloc(size_t nblocks,
size_t nbytes);
void upc_free(shared void*ptr);
```

`upc_global_alloc()` is a non-collective call (should be called by only a single thread) that allocates a contiguous memory region in the shared memory space equivalent to `shared[nbytes]char[nblocks*nbytes]`. If called by more than one thread, multiple regions of memory are allocated and each thread is returned a different pointer. `upc_free()` is also a non-collective call and frees dynamically allocated shared memory pointed to by `ptr`.

Check Out Other UPC Resources

UPC offers a number of other features, including synchroniza-

See Extreme, *pg. 56*

FIGURE THREE: An error results from having the `upc_barrier` after the loop containing `upc_memget()` instead of before it

```
[gcc_upc]$ upc -fupc-threads-2 mmmult.upc -o mmmult
[gcc_upc]$ ./mmmult
     +-                                        -+
     |  60.0  54.0  48.0   0.0   0.0   0.0|
c =  | 168.0 153.0 138.0   0.0   0.0   0.0|
     | 276.0 252.0 228.0 204.0 180.0 156.0|
     | 384.0 351.0 318.0 285.0 252.0 219.0|
     +-                                        -+
```

Printing the other matrices reveals the source of the problem:

```
     +-             -+
     |  1.0  2.0  3.0|
a =  |  4.0  5.0  6.0|
     |  7.0  8.0  9.0|
     | 10.0 11.0 12.0|
     +-             -+

     +-                                  -+
     | 18.0 17.0 16.0 15.0 14.0 13.0|
b =  | 12.0 11.0 10.0  9.0  8.0  7.0|
     |  6.0  5.0  4.0  3.0  2.0  1.0|
     +-                                  -+

           +-                                  -+
           | 18.0 17.0 16.0  0.0  0.0  0.0|
b_local =  | 12.0 11.0 10.0  0.0  0.0  0.0|
           |  6.0  5.0  4.0  0.0  0.0  0.0|
           +-                                  -+
```

Extreme, *from pg. 50*

tion locks, non-blocking/split-phase barriers, and other explicit communications calls, all of which make it a very powerful language for parallel programming. The strengths of UPC are its simplicity and its good performance for embarrassingly parallel problems; however, these additional language features make it possible to do almost anything that can be done with MPI (the Message Passing Interface) while minimizing the amount of message passing code that must be written.

FIGURE FOUR: The correct output for the matrix-matrix multiply program with the `upc_barrier` placed before the loop containing `upc_memget()` for *GCC UPC* and for Berkeley's implementation

```
[gcc_upc]$ upc -fupc-threads-2 mmmult.upc -o mmmult
[gcc_upc]$ ./mmmult
     +-                                      -+
     |  60.0  54.0  48.0  42.0  36.0  30.0|
c =  | 168.0 153.0 138.0 123.0 108.0  93.0|
     | 276.0 252.0 228.0 204.0 180.0 156.0|
     | 384.0 351.0 318.0 285.0 252.0 219.0|
     +-                                      -+

[bupc]$ upcc --network=udp -T=2 -o mmmult mmmult.upc
[bupc]$ upcrun ./mmmult
UPCR: UPC thread 0 of 2 on node1 (process 0 of 2, pid=7071)
UPCR: UPC thread 1 of 2 on node2 (process 1 of 2, pid=6375)
     +-                                      -+
     |  60.0  54.0  48.0  42.0  36.0  30.0|
c =  | 168.0 153.0 138.0 123.0 108.0  93.0|
     | 276.0 252.0 228.0 204.0 180.0 156.0|
     | 384.0 351.0 318.0 285.0 252.0 219.0|
     +-                                      -+
```

For many complex models, MPI still offers the best performance, but for image processing and a variety of simpler parallel computing tasks, UPC can offer equal performance and better code maintainability.

These three UPC tutorials should give you enough to get started in writing your own code in UPC. Additional tutorials and documentation can be found at the UPC Web sites at George Washington University (http://upc.gwu.edu/) and the Lawrence Berkeley National Laboratory (http://upc.nersc.gov/). Download one of the compilers and try it out for yourself!

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

Git, *from pg. 37*

```
Push: test:test
Push: dev:dev
```

There is no hard requirement that the same name be used on both sides of the refspec colon such as `test:test`. However, care should be taken to track the flow of objects and the expected use patterns carefully. If you're expecting to have other users `clone` and `pull` from this repository, it is quite likely, though not required, that you want to ensure that your push updates somehow land in the master branch. Furthermore, since refspecs can potentially match branch names, tags and heads, and so on, you can be more explicit if needed. For example:

```
Push: refs/heads/test:ref/heads/test
Push: refs/tags/today:ref/tags/today
```

If you're providing either HTTP protocol support for clones of your repositories, or a *gitweb* interface to them as described here, you should perform one more important change to your repositories to support them. Both of these tools rely on a few files of pre-built information located in the *.git/info* and *.git/objects/info* directories.

After a `git push` operation has updated a repository, the `git update-server-info` command must also be executed. While you may run this by hand from within the repository with the `GIT_DIR` environment variable set to ., it is easier and more reliable to enable the *post-update hook*. It's located in the *hooks* subdirectory, and can be enabled by simply making it executable. The `post-update` hook is automatically invoked on the remote repository after it has finished updating all of the references initiated by a local `git push` operation. The default action for the post-update hook is to perform the `git update-server-info` command!

Go Forth and Collaborate!

You now have all of the necessary tools to collaborate with *git*. You know how to create new *git* repositories, base new development off of a clone of some pre-existing public repository, set up a *git* server using HTTP and *git* protocols through Apache and basic Linux services, provide a web-based front-end for your repository, and finally, know how to create and update it. With that, other developers will be able to use and leverage your work.

There is still plenty of room to explore how best to provide development branches, publish works in progress, leverage distributed development efforts, and coordinate between different remote sites!

Jon Loeliger currently works at Freescale Semiconductor developing Linux for the PowerPC.