Unified Parallel C
UPC on HPCx

Ian Kirker and Adrian Jackson

January 14, 2008

Abstract

UPC is an alternative to MPI and OpenMP parallelisation. It is an extension of C that aims to simulate a shared memory environment, hiding the details of parallelisation from the user. This document outlines the basic concepts of UPC, and explores what functionality is available on HPCx. It then goes on to analyse the performance of UPC against IBM’s MPI and LAPI on HPCx. Both IBM’s UPC offering, and an open-source (Berkeley) UPC compiler are evaluated.

1 Introduction

UPC is an extension of C, designed to use message passing techniques to simulate a shared memory multiprocess environment. It has numerous features designed to make parallelisation as simple as possible, whilst also attempting to preserve the efficiency and overall structure of C. It provides a Partitioned Global Address Space for the transfer of data between processes, as well as numerous synchronisation and collective functions that enable the control of program flow between parallel threads.

This first sections of this document attempt to provide a brief overview of the features UPC provides, and their application and availability on HPCx. The later sections analyse the relative performance and capabilities of the Berkeley UPC distribution to the IBM MPI and LAPI libraries on the HPCx platform. To get the clearest picture of network performance, we use reduced ‘ping-pong’ codes, and run between two nodes of the HPCx Phase 3 installation. Collective IO functions are also tested, writing and reading strings from multiple nodes to a single file.

IBM provides an alpha version of a UPC compiler, XL UPC, which provides the basic functionality of UPC. However, it was created in 2005, and the UPC specification has been extended since that time, adding new collective routines and suggested IO functions. A collaboration of the Lawrence Berkeley National Laboratory and the Berkeley University of California has produced a cross-platform UPC distribution that works on HPCx however, and can be compiled if one first adds the GNU compiler, which it requires as a preprocessor.

UPC is designed to make it easy to learn from C, and to make the parallelisation of C programs an easy process. A large proportion of the UPC functionality acts to duplicate standard C abilities as applied to the shared memory context, making their use quite intuitive, although not necessarily as adaptable as they could be.

1.1 Berkeley UPC

The Berkeley UPC (Unified Parallel C) distribution is a cross-platform implementation of the UPC v1.2 specification with additional useful extensions. It uses a source-to-source translator to convert UPC code to C, and a choice of
compiler to then compile the C code. It can use a number of cross-platform and vendor-supplied networking protocols with the underlying GASNet library, including MPI, ELAN Quadrics, Infiniband and LAPI, however, HPCx only supports the use of LAPI and MPI. For performance reasons, it was elected to use the LAPI networking option for these tests, as well as using IBM’s xlc in 64-bit re-entrant modes as the underlying C compiler.

1.2 LAPI

IBM’s LAPI (Low-level Application Programming Interface) is the networking library supplied with their cluster technology for use in high speed message passing programming. It has one-sided transfer capabilities in both normal and RDMA methods, however, its relative complexity and lack of portability limit its use. For these tests, LAPI code was compiled using the IBM’s mpcc compiler in 64-bit re-entrant mode, which gives best performance and thread safety, while also ensuring its relevance to the method used for the UPC code.

1.3 MPI

The open specification for MPI (Message Passing Interface) has both open-source and vendor-supplied implementations, including IBM’s own. It is currently regarded as the basic standard for the majority of network-using message-passing programs. There are currently two active specifications for MPI, MPI-1.2 and MPI-2; MPI-1.2 contains basic point-to-point transfers, both synchronous and asynchronous, as well as collective data transfer routines and synchronisation features. MPI-2 extends this by adding one-sided communication, functions for changing the MPI environment mid-process, and collective IO features. The IBM MPI implementation currently implements MPI-1.2 and a number of features of MPI-2, including the one-sided communication and collective IO, but keeps the static environment of MPI-1.2. For these tests, MPI code was compiled using the IBM’s mpcc compiler in 64-bit reentrant mode, as with the LAPI code.

1.4 HPCx

The HPCx platform is described in great detail in other reports, but, in brief, the Phase 3 HPCx installation contains 160 compute nodes which each contain 16 1.5 GHz 64-bit Power5 processors, with each node being connected via two twin-linked network adapters to a High Performance Switch.

2 The Parallel Global Address Space

The Parallel Global Address Space (PGAS) is the simulated shared memory space that allows different threads to access the same data. Although accessible by every thread, every part of the data has an ‘affinity’ corresponding to the thread that actually has that data in its memory. For this reason, memory access times to some parts of the data will be quicker than others, as local memory remains much quicker to access for the local thread.

PGAS variables are created using the shared type qualifier:

```c
shared type variable;
shared [block_sizeopt] type variable [index];
shared [] type variable [index];
shared [*] type variable [index];
```

Every shared variable and every element of a shared array is said to have affinity to the thread where that element is stored, and all non-array shared variables have affinity to thread 0.
The block-size qualifier can be used to control the distribution of elements of an array among threads - by default, elements are serialised and divided such that the i-th element has affinity to thread i%THREADS). If block-size is specified, elements are serialised and divided such that the i-th element has affinity to thread:

\((i/block_size)%(THREADS*block_size)\)

If the special qualifier [ ] is given, all elements have affinity to thread 0. The special [*] qualifier is useful for multidimensional arrays - if given, the elements of the array are divided such that all of arr[i] has affinity to i % number_of_threads regardless of other dimensions.

Example:

```plaintext
shared  type array1 [4];
shared [2] type array2 [4];
shared []  type array3 [4];
shared [*] type array4 [4];

shared  type array5 [4][2];
shared [3] type array6 [4][2];
shared [*] type array7 [4][2];
```

```
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>array3[0..3]</td>
<td>array4[0]</td>
<td>array5[0]</td>
<td>array6[0]</td>
</tr>
<tr>
<td>array4[0]</td>
<td>array5[0][0]</td>
<td>array5[0][1]</td>
<td>array5[0][2]</td>
</tr>
<tr>
<td>array5[0][0]</td>
<td>array5[0][1]</td>
<td>array5[1][0]</td>
<td>array5[1][1]</td>
</tr>
<tr>
<td>array6[0][0]</td>
<td>array6[0][1]</td>
<td>array6[1][0]</td>
<td>array6[1][1]</td>
</tr>
<tr>
<td>array7[0][0]</td>
<td>array7[0][1]</td>
<td>array7[1][0]</td>
<td>array7[1][1]</td>
</tr>
<tr>
<td>array7[0][1]</td>
<td>array7[1][1]</td>
<td>array7[2][0]</td>
<td>array7[2][1]</td>
</tr>
</tbody>
</table>

Pointers can also be or point to shared variables, although there is a slight performance cost attached to dereferencing a shared variable, in addition to that of actually retrieving the variable’s value. Pointers that are set to an element of an array will also acquire a property called “phase” which tracks at what point the pointed to address sits in a block, and allows the pointer to be arithmetically operated on correctly.

Note that pointers do not acquire block size from the address they refer to - they may have block size set at declaration, otherwise there is no guarantee that a pointer will traverse array values correctly. Also note that shared variables cannot be declared within program flow or within structs - local pointers to shared can, however. Note that shared structs may not interact as expected with external functions - as the pointer to shared construct is only legitimate within UPC, the struct will only be externally accessible to non-UPC libraries from the thread it is associated with.

```plaintext
int  *ptr1  Local to local
shared int *ptr2 Local to shared
int  *shared ptr3 Shared to local (unusable)
shared int  *shared ptr4 Shared to shared (not available with XLUPC)
```
3 PGAS Library Functions

Included in the UPC library functions are analogues to the normal C string functions that act on strings in the shared address space, as well as some designed to act on and read the affinities of pointers and variables.

3.1 Strings

The string functions `upc_memput`, `upc_memget` and `upc_memcpy` work as one might expect - `upc_memput` copies a string from a local source to a shared destination, while `upc_memget` performs the opposite function, and `upc_memcpy` copies a string from a shared source to a shared destination. There is also a `memset` equivalent, `upc_memset`. Note that the shared destination written to must be entirely contained in a single thread. These functions are unsurprisingly much faster than an assignment loop at such operations due to the lack of referencing and message overhead.

```c
void upc_memcpy(shared void * restrict dest, shared const void *restrict src, size_t n);
void upc_memput(shared void * restrict dest, const void * restrict src, size_t n);
void upc_memget(void * restrict dest, shared const void * restrict src, size_t n);
void upc_memset(shared void *dest, int c, size_t n);
```

3.2 Memory

A thread can dynamically allocate shared memory using the `upc_alloc` function - however, memory allocated this way always has affinity to the calling thread. The `upc_global_alloc` and `upc_all_alloc` functions allow the dynamic allocation of shared space across threads, with the latter being a collective function that must be called from all threads, returning the same reference to shared to each.

```c
shared void *upc_alloc (size_t nbytes);
shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
shared void *upc_all_alloc (size_t nblocks, size_t nbytes);
```

Shared space allocated by `upc_global_alloc` or `upc_all_alloc` is distributed as would an array declared by:

```c
shared [numbytes] char [nblocks * numbytes]
```

Memory allocated by any of these methods may be freed by use of the `upc_free` function:

```c
void upc_free(shared void *ptr);
```
3.3 Strict vs Relaxed

UPC has two shared memory access models - strict and relaxed - which may be set by region of code by #pragma directives, or per variable by use of declaration qualifiers. Under relaxed mode, accesses may be reordered for speed, so care must be taken to prevent race conditions between threads, or parts of data being overwritten by simultaneous write accesses. Conversely, strict mode ensures that all accesses are absolutely ordered, and that each write is finished before the next may begin. Although relaxed mode accesses may be reordered, they may not move past a strict access. Additionally, the statement upc_fence exists as a null strict access, ensuring all accesses are complete. It should be noted that relaxed is the default behaviour, and that two header files exist, upc_relaxed.h and upc_strict.h which merely use the #pragma statement and then include upc.h.

```c
#pragma upc relaxed
#pragma upc strict

shared strict type variable;
shared relaxed type variable;
```

4 Parallelisation

4.1 Defined Constructs

Parallelisation in UPC can stem from the use of either the two thread monitoring expressions, MYTHREAD and THREADS, or the parallel loop statement, upc_forall.

THREADS - A macro giving the total number of threads. If the number of threads to use was defined at compilation, this may be used as an ordinary integer expression. Otherwise, it may only be used as or to multiply an integer constant alone. MYTHREAD - Integer expression, being the index of the current thread. Range is 0..THREADS-1.

upc_forall - This construct works in an equivalent fashion to a normal for loop, except that it automatically divides the loops among the threads according to a fourth parameter:

```
upc_forall( initial; test; increment; affinity )
```

The affinity parameter can be either a pointer, or an integer expression. If a pointer, loops will execute on which ever thread is local for the address or addresses referenced. If an integer expression, the loop will execute on thread i%THREADS. Because of the nature of this construct, modification of the variables used within during the loop is not recommended without careful forethought.

Below are some examples of code using the upc_forall function, and the values of variables as a result of the code execution.

Example 1:

```c
#include <upc_relaxed.h>

int main(int argc, char **argv)
{
    int i, numbers[10];
    upc_forall(i=0;i<10;i++, i)
    {
        numbers[i] = MYTHREAD;
    }
}```
Example 2:
#include <upc_relaxed.h>

int main(int argc, char **argv)
{
  int i;
  shared int numbers[10];
  upc_forall(i=0;i<10;i++, i)
  {
    numbers[i] = MYTHREAD;
  }
}

Example 3:
#include <upc_relaxed.h>

int main(int argc, char **argv)
{
  int i;
  shared [5] int numbers[10];
  upc_forall(i=0;i<10;i++, i)
```c
#include <upc_relaxed.h>

int main(int argc, char **argv)
{
    int i;
    shared [5] int numbers[10];
    upc_forall(i=0;i<10;i++, &numbers[i])
    {
        numbers[i] = MYTHREAD;
    }
}
```

Example 4:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations Executed</td>
<td>Iterations Executed</td>
<td>Iterations Executed</td>
<td>Iterations Executed</td>
</tr>
<tr>
<td>i = 0</td>
<td>i = 1</td>
<td>i = 2</td>
<td>i = 3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values of numbers</td>
<td>Values of numbers</td>
<td>Values of numbers</td>
<td>Values of numbers</td>
</tr>
</tbody>
</table>

5 Thread Synchronisation

There are three different mechanics for synchronisation: barriers, locks, and fences.
5.1 Barriers

UPC has two different barrier methods: ordinary, and ‘split-phase’ barriers. The upc_barrier statement merely pauses execution until all threads have reached a barrier statement, while the upc_notify/upc_wait combination allows the thread to perform additional operations while waiting. Barriers may have labels to test for the correct point - if one thread reaches a barrier while another thread has reached a differently labelled barrier, an error occurs and execution ends.

upc_barrier; Blocks until all threads reach this point.
upc_barrier label;

upc_notify; Note that thread has reached this point,
upc_notify label; continue until a wait statement

upc_wait; Wait until other threads have reached
upc_wait label; notification point.

5.2 Locks

Locks allow the marking of sections of code such that only one thread may execute them at one time. Using a lock requires the use of an opaque type, upc_lock_t, and one of several lock functions:

5.2.1 Allocation

upc_lock_t *upc_all_lock_alloc(void);
upc_lock_t *upc_global_lock_alloc(void);

void upc_lock_free(upc_lock_t *ptr);

A lock variable has two states - locked and unlocked. Newly allocated locks are always unlocked. upc_global_lock_alloc allocates a lock and returns a pointer to it. upc_all_lock_alloc is a collective function - it is called by every thread, and every thread receives the pointer to the same lock object. upc_lock_free deallocates a lock created by either of these functions, regardless of status.

5.2.2 Use

void upc_lock(upc_lock_t *ptr);
int upc_lock_attempt(upc_lock_t *ptr);
void upc_unlock(upc_lock_t *ptr);

The upc_lock function has two effects - if the lock struct is already locked, execution waits until the lock struct is unlocked. If the lock struct passed is unlocked, its status is set to locked. upc_lock_attempt does not wait, instead returning 0 if the lock is already set. Attempting to call either of these functions upon a lock that has already been set within the same thread has undefined results. upc_unlock sets the state of the struct passed to unlocked, however, this behaviour is only specified for the thread that initiated the lock.

Example:
#include <stdio.h>
#include <upc_relaxed.h>
int main(int argc, char **argv) {
    int i;
    for (i=0; i<5000; i++)
        printf("%c", 'a'+MYTHREAD);
    return 0;
}

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

int main(int argc, char **argv) {
    int i;
    upc_lock_t lock;
    upc_lock(lock);
    for (i=0; i<5000; i++)
        printf("%c", 'a'+MYTHREAD);
    printf("\n");
    upc_unlock(lock);
    return 0;
}

Output: cccccc...bbbbbabb...aaaaaaa...

5.3 Fences

Fences synchronise the shared memory accesses of a thread. The statement `upc_fence` is a null strict access, ensuring all accesses are complete once the statement has completed.

5.4 Collective Functions

There are three groups of collective functions defined within the UPC 1.2 specification -
- memory allocation and locking functions, defined in `upc.h` and mentioned above
- mass memory functions for gathering and scattering, defined in `upc_collective.h`
- collective IO functions, defined in `upc_io.h`

The collective IO functions are currently defined as an optional extension in implementations. The Berkeley UPC distribution currently implements all of these groups, while IBM's only implements the allocation and locking functions.

Collective functions must be called by all threads simultaneously and, as such, imply a barrier before and/or after the call. The names of collective functions in UPC, by convention, begin with `upc_all_`. 
6 Implementation Specifics

6.1 Compilation

Compilation of UPC code may proceed through first translating and converting to C, then normal C compilation. The Berkeley UPC distribution has the option of using a remote translator available through either their online services, or as C++ source. The translator is currently supported on fewer platforms than the runtime/compilation package - on HPCx it will only compile with GCC.

Currently, the BUPC compiler may use the XLC compiler on HPCx to actually compile the converted C code, but requires the GCC preprocessor for the translation and conversion process. The compiler used must be specified at compilation into the upcc converter using the configure variables like so: ./configure CC="xlc_r -q64" CXX="xlC_r -q64" MPICC="mpcc_r -q64"

Use of the re-entrant versions and the 64-bit switch is recommended for optimal thread-safe programs. Note that the IBM UPC compiler compiles as 32-bit by default (-q64 to switch to 64-bit).

The alpha IBM UPC distribution and the GCC UPC compiler also provided by the BUPC team compile UPC code ‘directly’, but there is currently no version of the GCC UPC compiler for powerpc processors.

The number of threads to use may be set at compile time by options passed to the compiler:

upcc -fthreads 4
xlupc -qupcthreads=4

Also note that while the Berkeley UPC translator will accept a file with any extension as UPC, the IBM UPC compiler demands that UPC programs have the .upc file extension.

The Berkeley UPC compiler accepts object files compiled in other languages, as long as they provide a C-compatible interface - for example, Fortran and C++ can both be used with UPC, as with a normal C program. If the main function is not located within UPC code, however, UPC must be bootstrapped and manually exited using UPC extension functions bupc_init_reentrant(), which should be the first statement in the main function, and bupc_exit(), which should be the last. The extension libraries included with the BUPC distribution also include a number of functions to make the inclusion of C++ and C source easier, including functions to allocate shared memory from non-UPC code, which another extension, the bupc_local_to_shared function can then use to retrieve a pointer to shared. Object files capable of being used in this way include one peculiar addition, MPI programs compiled with mpcc. Programs compiled in this way seem to attain threads that are both UPC and MPI capable, rather than the more obvious result of obtaining a number of MPI threads for each UPC thread. Passing data between UPC and MPI functions in such programs must be done using strict C interfaces - obviously the MPI compiled objects will not accept or dereference shared pointers, nor Ivalues for shared variables.

The IBM UPC compiler can similarly compile in object files created by other languages, however, it is unclear what features are available for use in such circumstances. It seems that the main function must be in UPC code, however.

6.2 Networks

The IBM UPC implementation is based on pthreads - it cannot run across a network. Conversely, the Berkeley UPC library allows for a wide choice of underlying network choices - on HPCx, it allows for the use of LAPI or MPI as the underlying network protocol with the option of also using pthreads within these, or instead of any networking.

With a compatible MPI compiler, the BUPC translator can also be compiled to be capable of compiling mixed MPI/UPC programs (in the same source file), even if MPI is not the underlying network for the UPC transmissions. Unfortunately, due to the rather unusual nature of the mpcc compiler, attempting to use this feature on HPCx proved more of a challenge than could be overcome.
6.3 Executing

Executables created by the IBM’s compiler can be run from the command line, as they do not use any networking protocols. Berkeley UPC executables use the system’s network libraries and thus must be run using poe and network settings appropriate to whichever protocol UPC is using. The number of threads may be set in the loadleveler file by setting the **UPC_NTHREADS** environment variable.

Example UPC loadleveler file:

```bash
#@ shell = /bin/ksh
#@ job_name = upc_test
#@ job_type = parallel
#@ tasks_per_node = 65
#@ cpus = 65
#@ node_usage = not_shared
#@ network.LAPI = csss,shared
#@ bulkxfer = yes
#@ wall_clock_limit = 00:15:00
#@ account_no = z004-ssp
#
#@output = $(job_name).$(schedd_host).$(jobid).out
#@ error = $(job_name).$(schedd_host).$(jobid).err
#@ notification = never
#@ queue

UPC_NTHREADS=65
echo ---------------
env | grep UPC
echo ---------------

poe ./upc_test
```

6.4 Debugging

Berkeley UPC programs can, theoretically, be debugged by Etnus’ TotalView. Support must be compiled into the upcc compiler, however, and neither gcc nor xlc would allow the totalview debugging-enabled translator to configure correctly.

Ordinary C debuggers can also be attached to individual threads, and there are a number of options to support this involving suspend points and barriers requiring external input to release, but this requires login access to the individual nodes, and is thus not supported on HPCx.

The BUPC translator can be compiled with the **-enable-trace** tag, which allows UPC programs made with it to be run with network tracing enabled. These tracefiles can then be interpreted by a tool called **gasnet_trace**, which allows the viewing of all network activity during the run of the program.

7 Data Transfer

One of the large benefits of UPC is that the assignment operator can be used to carry out transparent one-sided put and get operations. However, this is not efficient over loops, as each operation becomes an atomic set of dereference, check locality, and retrieve data operations. Therefore, the UPC libraries provide the memput, memget
and memcpy functions to copy strings of data from and to shared memory. Though more complex to effect, these appear to be significantly faster than the inline assignment transfers.

![Figure 1: Fast Data Transfer Method Times](image1.png) ![Figure 2: Slower Data Transfer Method Times](image2.png)

For the slower methods, comparisons were made using mean values from trials of 100 iterations of each transmission method, each iteration consisting of a message going from the master thread to the slave thread on a different LPAR, and then being returned. The faster methods were tested using a similar method, but with 10 trials of each set of 100, taking the fastest value from a single thread with 15 others performing simultaneous transmissions.

From these figures, we can see that the UPC assignment operator, functions as expected for large quantities of data, and is relatively slow compared to other methods. The memput function is comparable at least in scale of speed terms to LAPI and MPI, but seems to have higher overheads, in both computational and network terms. The progression from a mean higher speed to a mean slower speed is suspected to be a function of the underlying network libraries, and its earlier occurrence in the UPC memput figures would seem to indicate that the network overhead approximately 10 times that of the LAPI and MPI methods. The memget function seems to have a similar transfer time to memput, but suffers the progression at a similar point to the faster methods, and to a lesser degree. This suggests that memget has lower overheads, possibly due to the lower address transmission requirements inherent to the pull method in this case.

The MPI2 figures appear to be quite erratic, and this was borne out by repeated tests. Further cycles could have been calculated to get more figures for a mean, but this method was already particularly slow.

### 8 I/O Functions

The latest UPC specification (1.2) proposes a collective I/O library designed in a similar fashion to those of the MPI libraries. The Berkeley UPC distribution currently contains a reference implementation of these extensions currently, though a fully optimised production implementation is planned. This reference implementation channels all the I/O through one thread, so these functions may not stand up as stably in a high-throughput program across many nodes with simultaneous high-use data transmission.

Comparisons were made between the basic write and read operations of the UPC and MPI’s own I/O functions, taking the mean time of 100 writes and reads of increasing sizes with 16 threads working on the same file simultaneously. The comparative data transfer rates show that the UPC I/O functions seem to be much more efficient, though still nowhere near as fast as native POSIX I/O, which gives read and write times in the single microseconds.

In the plotting of write speed, the MPI non-collective and collective operations have not been differentiated between - this is because the difference in performance was negligible on the scale of the difference between them and the UPC functions’ performance.
A preliminary test was also carried out on how the number of nodes operating simultaneously affected performance of the UPC library, writing 1, 10, and 100 bytes to the same file, and then reading them back. The results indicate that although the time taken per byte does increase with the number of nodes operating simultaneously, even at 512 nodes, the time is still manageable. For more sporadic data transmission paradigms, though, a dedicated or merely specified IO thread using ordinary C IO functions would probably be more efficient.

9 Conclusions

The Berkeley UPC distribution seems to provide all the features needed to compete with standards designed to fulfil the same purposes - on HPCx, mature libraries implementing the LAPI, OpenMP and MPI libraries for parallel applications are currently provided by IBM. Providing easily usable one-sided communication, exceptionally convenient small transfers, and simple syntax that mirrors that of C, UPC appears simpler to use than LAPI or MPI, but lacking in some diagnostic and advanced error checking that may be necessary in more advanced applications.

Despite the XLUPC distribution’s limitations, it is still suitable for small scale parallelisation, in a situation where OpenMP might normally be used. A new version is being planned by the IBM developer team to be released with the next version of XLC, but this will also lack network support. However, this next version may include some of the more recent features, which should make it stand above OpenMP in terms of features. There are no firm plans for a version that is LAPI-capable yet, but the development team at IBM has claimed some interest in the idea.

Berkeley UPC methods fall into one of two groups - one group are simple and transparent, but slow, data transmission method; the other group provides faster and slightly more complex that compares to the other high calibre examples. These used together could be especially useful, as in the instance where a single short message must be used to transmit information used to receive a larger transmission.
The fact that BUPC’s data transmission methods (as well as parallelisation and work division constructs) are so similar to their equivalent ordinary C equivalents combined with their reasonable performance would seem to indicate that it would be exceptionally useful for the task of rapid parallelisation porting - taking a serial source and quickly converting it to run in a parallel manner. The high performance of its collective IO would also suggest its use in programs requiring a high degree of file interaction over MPI.

Although it has lower high-speed performance than either LAPI or MPI on this system, its performance is good enough and BUPC’s portability high enough that UPC code may also be a good option where the code must be portably fast to other high-performance platforms.

References


All links are correct and responsive at 6/9/07.