An Evaluation of UPC in the Ludwig Application

Alan Gray
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

April 29, 2009

Abstract

Unified Parallel C (UPC) is an extension to the C programming language offering the use of shared data structures to facilitate parallel programming and potentially improve performance, and is fully supported on modern systems such as the Cray X2. The introduction of UPC functionality to key sections of the Ludwig Lattice Boltzmann application is seen to reduce complexity by circumventing the need for data structure halo cells, and associated message passing halo-swap routines. A straightforward adaptation involving direct use of UPC shared structures is observed to have a substantial negative impact on performance, mainly attributable to shared pointer operation overheads. Optimisation involving use of regular C pointers (obtained from casting) where possible is found to largely eliminate the performance gap, with the remaining difference likely attributable to overheads concerning the remaining necessary shared array accesses.

KEYWORDS: UPC, Ludwig, Lattice Boltzmann, X2

1 Introduction

Modern HPC architectures typically comprise multiple units connected together via a high performing interconnect, where each unit, or *node*, contains one or several processing cores and memory. Applications which wish to utilise multiple nodes to solve a single problem must incorporate a mechanism for each process to acquire remote data. The Message Passing Interface (MPI) library, which has become the de-facto standard, allows data to be transferred between nodes via the sending and receiving of packets of data (messages). This has proved to be an extremely effective solution but is not without caveats, notably

- the need for complex coding to manage the message passing
- performance overheads associated with the underlying 2-way communication

The advent of novel Partitioned Global Address Space (PGAS) programming methods address these issues. At the application level, PGAS languages allow remote data to be accessed in a similar way to local data, avoiding the need for message passing. This allows for the minimisation of code complexity and performance overheads (but the latter depends on the hardware used and associated PGAS implementation).

Chapel, X10, Fortress and Titanium are all new PGAS languages currently in a developmental stage. The former 3 were reviewed in an HPCx technical report [1], and the latter in a recent EPCC MSc dissertation [2]. Unified Parallel C (UPC) and Co-Array Fortran (CAF) are extensions to the C and Fortran languages which incorporate PGAS methods, and full implementations are currently available. In particular, UPC and CAF are supported at the compiler and hardware level on current Cray vector systems. UPC was chosen as the subject of this study over CAF for reasons of portability: CAF is currently only supported by a handful of compilers (most notably on Cray systems), whereas UPC is available on a range of systems (as discussed in section 2.1.1).

For development of new HPC applications, or parallelisation of existing serial applications, the choice of UPC over MPI will avoid the need to manage the passing of messages and therefore will reduce the development effort. However, there are many applications which already use MPI. For these, a conversion to UPC may be beneficial for both performance and maintenance reasons.

This study investigates the conversion of an existing application, the Ludwig Lattice Boltzmann code, from MPI to UPC. It was beyond the scope of the study to perform a full conversion, but the ability of UPC and MPI to co-exist in the same application allows for a conversion of key sections, giving reliable indications as to how a fully converted UPC code would compare to the original MPI version, in terms of both complexity and performance.

Section 2 gives an introduction to UPC and the Ludwig Application. The conversion work undertaken and resulting performance comparisons are presented in Section 3.

2 Background

2.1 UPC

An introduction to UPC was given in an HPCx technical report [3]. This section recaps some basic concepts in the context of the current work.

Note that the term `process` will be used to refer to a single task (e.g. MPI task or UPC thread) with associated local memory. The number of processes per physical node will depend on the architecture, but is generally not relevant to the discussion.

The main feature offered by UPC is the ability to use a single data structure, declared using the `shared` qualifier, which is distributed (in terms of physical storage) between processes but fully accessible by any process. As an example, consider an 8 element integer array to be decomposed between 2 processes. The standard technique would be to declare a 4-element array private to each of the processes:

```
int p[4];
```

In order for a process to access non-local data, an additional data structure or extension of `p`, along with calls to a message passing library, are required. If it is sufficient to have access to only neighbouring cells (which is commonly the case), *halo cells* can be added to the boundaries of the array, which becomes

```
int p[6];
```

where `p[1]` to `p[4]` contain the *interior* data and the halo cells `p[0]` and `p[5]` (assuming periodic boundary conditions) contain a copy of remote data obtained via "Halo Swapping" routines containing MPI calls.

The alternative UPC declaration

```
shared [8/THREADS] int s[8];
```

creates an array for which data storage is physically distributed between the 2 processes but is indexed globally (and thus has 8 elements rather than 4) such that any element can be accessed by any process. The THREADS variable is automatically set to the number of UPC threads (i.e. 2 in this case): the `[8/THREADS]` part of the declaration instructs that a regular decomposition is performed (with the first half of the array distributed to the first process, and the second half to the second). An access to `s[0]` would, if made on the first process, involve a local memory access but if made on the second process involve a remote memory access.

Figure 1 gives an illustration of these private and shared data structures, where halo cells have been added to the private arrays. It is clear that the shared structure offers a substantial reduction in programming complexity, since direct remote memory accesses remove the need for both halo cells and routines to pass messages containing remote data.

2.1.1 Platforms

Availability and performance of UPC is extremely dependent on the platform and compiler. Use of UPC involves, at the application level, direct accessing of remote data. One way for a compiler to implement this is to translate the UPC to calls to a message layer existing on the system software stack. For example, on IBM machines such as HPCx [7], available compilers using this technique [3] include the proprietary (alpha version) XLUPC (which translates to LAPI) [4] and the cross platform Berkeley UPC [5] (which can translate to LAPI or MPI). The latter is available across a range of platforms (and can translate to a number of network layers).

New architectures, however, such as the Cray X2, have global addressing capabilities and as such support languages such as UPC directly without the need for translation to message layers [6]. Clearly, this eliminates overheads. UPC code is compiled by the standard C compiler. Such support is expected to become more widespread in future architectures.

The X2 component of the HECToR service [8], containing 112 vector processors, was used for the developments and performance analysis contained in this paper[1].

[1]It should be noted that the fact that this architecture has vector rather than scalar processors is irrelevant here (since the study is concerned with communication rather than serial performance).

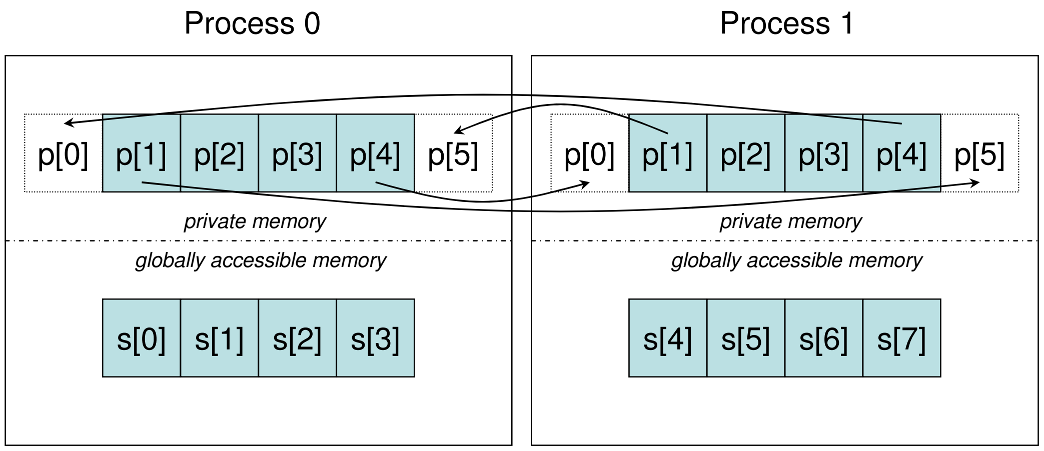

Figure 1: A conceptual layout of private and shared array data in memory for a 1-D array which has 8 elements of data distributed between 2 processes. Each process has a locally addressed private array p[] located in private memory, and the arrows represent data transfer from boundary interior elements to halo cells during halo-swapping. The globally addressed shared array s[] has no halo cells since direct memory access is possible to globally accessible memory.

2.2 The Application: Ludwig

The LUDWIG application, currently active in research, uses Lattice-Botzmann models to enable the simulation of the hydrodynamics of complex fluids in 3-D [9]. A comprehensive overview of the code, and Lattice Boltzmann methods in general, is given in a recent EPCC MSc Dissertation [10].

The main data structure in Ludwig is a 3-D lattice which is decomposed between the processes, such that each process has a local sub-lattice. The actual decomposition depends on the number of processes; for example, by default a 128^3 lattice is decomposed in all 3 dimensions on 64 processes (such that the local sub-lattice size is 32^3), but only decomposed in 2 dimensions on 4 processes (such that the local sub-lattice size is $64^2 \times 128$). The 3-D sub-lattice is linearised into an array named `site`, where each element contains a struct of type `Site`, containing physical variables[2]. The `site` array also contains elements corresponding to halo cells in each of the X,Y and Z directions, which are populated with edge data from neighbouring processes (in a similar fashion to the simplified illustration in Figure 1).

The left of Figure 2 provides a simplified illustration of the code's operation. Included in the initialisation stage is the dynamic allocation of the `site` array. For each timestep, the bulk of computation resides in the `Phi Gradients`, `Collision` and `Propagation` stages. The `Phi Gradients` and `Collision` stages involve only local on-site data operations (i.e. no halo cells of site are accessed). The `Propagation` stage involves non-local data operations: each lattice site is updated based on data from neighbouring sites, therefore a `Halo Swap` routine (which uses MPI) is needed to update the halo cells in advance of this section. The `Halo Swap` accounts for the vast majority of the application's communication overhead, but it should be noted that MPI is also used in several other places (including management of I/O).

Figure 3 shows a timing breakdown for the main timestep loop for a range of processing core counts. It can be seen that the `Collision` stage is most dominant, followed by the `Propagation` and `Phi Gradient` stages which contribute roughly equally in terms of computation time. The percentage contribution from the `Halo Swap` stage is seen to increase with increasing core count, as expected.

3 Integration of UPC into Ludwig

A full conversion from MPI to UPC was beyond the scope of this work, but the ability of UPC to coexist with MPI in the application allows for a gradual conversion, and in particular it is possible to perform investigative tests by converting only critical sections of the code: the `Propagation` section in this case (as discussed

[2]The Site structure contains a velocity density value for each forward and backward X, Y and Z direction, for each of two fluids.

```
--------------------
initialisation

loop over timesteps
  Phi Gradients
  Collision
  Halo Swap
  Propagation
end loop

finalisation
--------------------
```

```
--------------------
initialisation (including creation of
       local->shared mapping table)

loop over timesteps

  Phi Gradients
  Collision
  //Halo Swap no longer required
  copy local array to shared array
  Propagation (using shared array)
  copy shared array to local array

end loop

finalisation
--------------------
```

Figure 2: Pseudocode illustration of the original (left) and modified (right) Ludwig Application.

in the previous section). The strategy was to introduce a new shared data structure for use in `Propagation`, whilst maintaining the standard private structures and existing MPI functionality such that the converted section can be tested as part of a fully functioning application. In order to do this, it was necessary to develop functionality to map indices and copy data between the private and shared structures.

Firstly, a new UPC shared array to mirror the `site` data was created:

```
shared [SVOL/THREADS] Site s_site[SVOL];
```

where `SVOL` is defined to be the total volume (across all processes) needed to store all *interior* `Site` data (noting that halo cells are not required). Each process has affinity to `SVOL/THREADS` elements corresponding to the equivalent `site` private sub-lattice on that process (again neglecting halo cells)[3].

It was then required to create functionality for copying data between `site` and `s_site`. Firstly, a routine was created to map the (local) indices of each private `site` array to the corresponding (global) index of the `s_site` array. To do this, it is necessary to know the process ID on which the `site` element is resident, and local offset. For interior `site` indices, the mapping is fairly straightforward since the corresponding `s_site` element will have affinity to the same process. However, the halo `site` elements correspond to `s_site` elements resident on remote processes, requiring knowledge of neighbouring process IDs.

To illustrate, in reference to the simplified case in Figure 1, the mapping would be

```
sindex = local_interior_size*process_ID
          + index - halo_size
```

where `sindex` and `index` are the shared (`s_site`) and private (`site`) indices respectively, and `local_interior_size=4` while `halo_size=1` in this case. For interior `site` indices, the `process_ID` is simply the local process ID. For example, `index=1` on `process_ID=0` gives `sindex=0` while `index=1` on `process_ID=1` gives `sindex=4`.

For halo `site` indices, however, `process_ID` corresponds to the neighbouring rather than local process, e.g. the halo `p[5]` site on `process_ID=0` corresponds to the shared `s[4]` element, which is resident on process 1.

The mapping was developed for the more complicated 3-D case, where existing MPI functionality within the code was used to obtain neighbouring process IDs. For efficiency, this routine was used to create

[3]It is worth noting that the original code offers flexibility in that the lattice volume is read in from an input file at runtime, with the sub-lattice volume subsequently determined using MPI calls and the allocation done dynamically. This allows for a single executable which can be used for a variable number of processes and lattice sizes. In contrast, this UPC version requires the lattice volume SVOL to be hardwired in the code, and the number of THREADS declared on the compile line. This may be overcome using shared pointers and shared memory allocation functions [11].

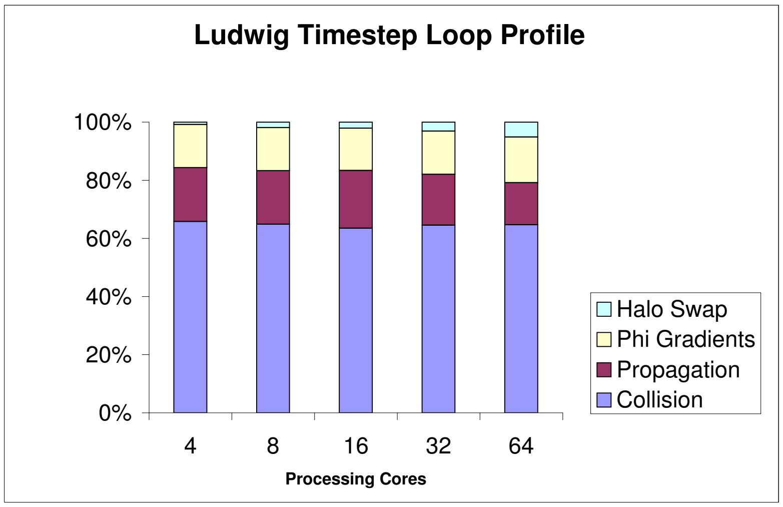

Figure 3: A profile of the main stages of the ludwig timestep loop displayed as a percentage of total loop runtime over a range of processing core counts.

an lookup table array named `sindex[]`. This then allowed the development of `get_site_from_shared()` and `put_site_in_shared()` routines to copy data between the `site` and `s_site` arrays. This indice mapping and data copying functionality allows for the `Propagation` code section to operate using the shared structure (and direct remote memory accesses) rather than the local structures (with halo cells and halo swaps).

To illustrate the conversion, consider an update to part of the `site` structure. The original code resembled

```
loop over index
 site[index].f[0] = site[index-1].f[0]+...;
```

To use the equivalent shared structure, the code is modified in a simple manner:

```
loop over index
 s_site[sindex[index]].f[0]
      =s_site[sindex[index-1]].f[0]+...;
```

which must then be sandwiched between `put_site_in_shared()` and `get_site_from_shared()` calls to integrate with the rest of the application.

The `Propagation` section was converted in this manner[4]. The adapted application can be illustrated as in the right of Figure 2. The qualatative benefits of UPC are immediately clear: the message passing `Halo Swap` section is not needed, and thus complexity is reduced[5].

The original and adapted codes were run across a range of processing core counts, with the timings of the `Propagation` section, plus associated communication, recorded. For the UPC code, all communication is performed directly within Propagation, but for MPI the communication is done externally during the Halo Swap section. Therefore, comparing the combined timing for `Propagation` plus `Halo Swap` (original code) verses `Propagation` alone (UPC) gives a reliable indication of how a full UPC code would perform in comparison to the original MPI version.

It is seen, from Figure 4, by comparing the (unoptimised) UPC performance results against the original MPI (Propagation plus communication) results, that the overhead in `Propagation` from use of UPC shared structure is considerably higher than use of the private arrays plus MPI halo swaps (across a range of core counts): the original performance is up to a factor of 1.8 higher than the

[4]In practice, it was possible to avoid the get_site_from_shared() call by actually using the site structure on the left hand side of the calls, since in this case all non-local accesses occur on the right hand side.

[5]Of course, as discussed this version includes additional functionality to copy data between local and shared arrays, and map corresponding indices etc. However, this added complexity is would not be required in a version of the code which had been fully converted to UPC.

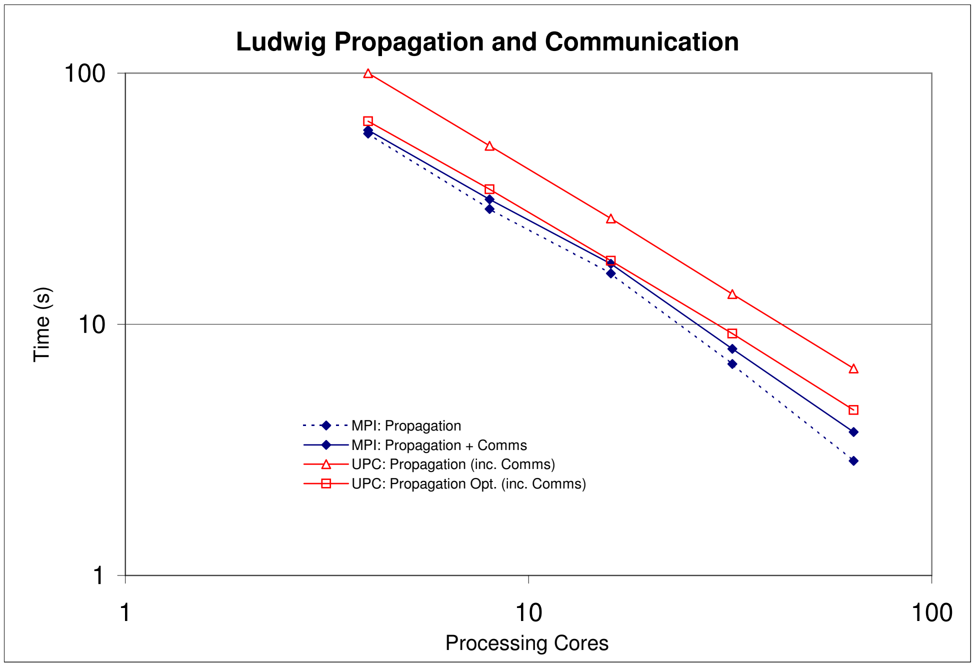

Figure 4: The time taken for the propagation section of the Ludwig code plus associated communication. For the MPI version (diamonds), the communication is done during a separate halo swap routine, whereas for the UPC version the communication is done directly in the propagation section. Shown are results for a UPC version where accesses are done through direct indexing of shared arrays (triangles), and an optimised version using regular pointers where possible (squares).

adapted UPC code for this section.

As already discussed, the one-sided remote memory access communications underlying UPC are supported at the hardware level on this system whereby MPI requires 2-way communication, so communication is an unlikely candidate for this UPC overhead. Indeed, it was found that the performance difference is instead largely attributable to the pointer operations involved with direct access of shared array elements. UPC Shared pointers internally differ from regular C pointers, and are typically more costly to dereference. An optimised version of the UPC code was created such that the memory accesses were performed using regular C pointers, obtained by casting from the shared pointers, for those updates (of non-boundary sites) which do not depend on remote data. Boundary updates require remote data through shared array indexing, as before. This adds complexity to the code, but as is shown in Figure 4 such an optimisation vastly reduces the performance difference. The MPI version still outperforms the optimised UPC version by a factor of 1.03-1.2: it is expected that this is due to the remaining shared pointer operation overheads.

4 Conclusions

Novel PGAS programming paradigms aim to address the complexity and performance issues relating to applications on modern massively parallel systems. UPC, a PGAS extension of C, which is comparatively mature and portable, offers the use of shared data structures allowing a more simplistic mechanism to access remote data than the current de-facto MPI library. UPC functionality was added to the Ludwig Lattice Boltzmann application. Although a full conversion was beyond the scope of the work, the ability of UPC and MPI to coexist allowed for the key section of the code to be converted to use UPC rather than MPI communications, and performance comparisons to be made. The use of UPC reduced complexity by allowing data structure halo cells and associated message passing halo-swap routines to be replaced by more intuitive direct remote memory accesses. A straightforward adaptation involving direct use of UPC shared data structures was found to perform significantly worse than the MPI version, but it was found that this was not primarily attributable to communication performance degradation, but instead to overheads involving shared pointer operations. An optimised version using regular C pointers (obtained via casting) where possible was found to perform more comparably to, but still slightly worse than, the MPI version. It is expected that the remaining performance difference is due to overheads with pointer operations involving the remaining (necessary) shared array accesses. It is clear that the Ludwig application is very sensitive to this issue, at least at the modest core counts available to this study; it would be interesting to obtain results for larger core counts.

5 Acknowledgements

Kevin Stratford (EPCC) provided access to, and useful discussions regarding, the Ludwig Application. Jason Beech-Brandt (Cray) provided helpful assistance and suggestions regarding the optimisation of the UPC code.

References

[1] "Chapel, Fortress and X10: Novel Languages for HPC", M. Weiland, HPCx Technical Report 0706, 2007, http://www.hpcx.ac.uk/research/hpc/ technical_reports/HPCxTR0706.pdf

[2] "Parallel Programming in Titanium An Evaluation of Usability and Performance", Florian Scharinger, EPCC MSc Dissertation, 2007, Available from http://www.epcc.ed.ac.uk/msc/dissertations/2006-2007/

[3] "Unified Parallel C: UPC on HPCx", Ian Kirker, Adrian Jackson, HPCx Technical Report 0709, 2007, http://www.hpcx.ac.uk/research/hpc/ technical_reports/HPCxTR0709.pdf

[4] IBM XLUPC compiler documentation, http://www.alphaworks.ibm.com/tech/upccompiler

[5] Berkely UPC web site, http://upc.lbl.gov/

[6] "The Cray BlackWidow: A Highly Scalable Vector Multiprocessor", Denis Abts et at., Supercomputing 2007 paper, http://sc07.supercomputing.org/schedule/ pdf/pap392.pdf

[7] HPCx service website, http://www.hpcx.ac.uk

[8] HECToR service website, http://www.hector.ac.uk

[9] Ludwig - A general purpose Lattice-Boltzmann code on the Cray T3E J. C. Desplat et al. http://citeseer.ist.psu.edu/411388.html

[10] "Message-passing for Lattice Boltzmann", Erlend Davidson, EPCC MSc Dissertation, 2008, Available from http://www.epcc.ed.ac.uk/msc/dissertations/2007-2008

[11] "UPC Language Specification, v1.2, including UPC collectives and UPC-IO libraries", available from http://upc.lbl.gov/docs/