Scientific Programming 18 (2010) 183–191
DOI 10.3233/SPR-2010-0310
IOS Press

Optimizing UPC programs for multi-core systems

Yili Zheng
Lawrence Berkeley National Laboratory, Berkeley, CA, USA
E-mail: yzheng@lbl.gov

Abstract. The Partitioned Global Address Space (PGAS) model of Unified Parallel C (UPC) can help users express and manage application data locality on non-uniform memory access (NUMA) multi-core shared-memory systems to get good performance. First, we describe several UPC program optimization techniques that are important to achieving good performance on NUMA multi-core computers with examples and quantitative performance results. Second, we use two numerical computing kernels, parallel matrix–matrix multiplication and parallel 3-D FFT, to demonstrate the end-to-end development and optimization for UPC applications. Our results show that the optimized UPC programs achieve very good and scalable performance on current multi-core systems and can even outperform vendor-optimized libraries in some cases.

Keywords: UPC, PGAS

1. Introduction

Multi-core processors have become mainstream: they are in almost all types of computing devices from commodity laptops to customized supercomputers. However, getting good application performance on multi-core systems is non-trivial except for those embarrassingly parallel programs. Many application domain research projects have shown that it requires significant amount of optimization and tuning to get high performance beyond simply parallelizing the code. One of the key challenges is to manage data locality well, which is crucial to achieving high performance on modern micro-processors with deep cache hierarchies and non-uniform memory access (NUMA) property.

NUMA optimizations are very important to get performance speed-ups [8,18]. However, it is difficult to handle NUMA issues when using a flat shared-memory model with threads (e.g., Pthreads and OpenMP) because little or no data locality information is available to the user. For example, OpenMP provides compiler directives to easily parallelize *for* loops but the speedups may be dismal if the data distribution and access locality are not optimized accordingly. The operating system may be able to alleviate the problem by using the "first-touch" memory allocation policy but the user has no way to query the data location subsequently. Moreover, the serial sections in parallel programs using the fork-join model can easily become the performance scalability bottleneck due to Amdahl's law.

PGAS programming models provide programming convenience similar to shared-memory programming models and at the same time enable users to manage data locality explicitly. Unified Parallel C is an C99 language extension with PGAS support and has become mature with production-quality implementations and supporting tools after more than a decade of research and development efforts. Berkeley UPC and GCC UPC are two active open source UPC implementations. Commercial offerings of UPC include Cray UPC, HP UPC, IBM UPC and SGI UPC. In addition, there are several research compiler infrastructures that support UPC, such as ROSE [14] and OpenUH [13]. Both IBM UPC [1] and Berkeley UPC [11] have demonstrated performance scalability on tens of thousands cores.

Though UPC and other PGAS languages were initially focused on large scale distributed-memory machines, they are also a good fit for emerging multi-core systems because the data partitioning capability of PGAS programming models helps users manage data locality efficiently and therefore achieve high performance. Figure 1 illustrates the PGAS model in UPC. The data partitioning information provided in the application programs can be used by the compiler and runtime to optimize the shared-data layout for NUMA.

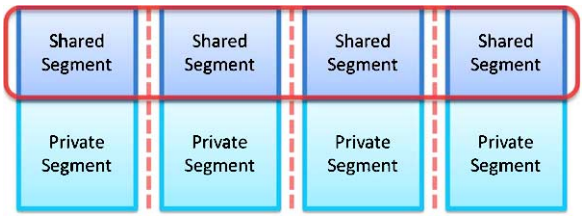

Fig. 1. Address space partitioning in UPC: vertical boundaries distinguish local and remote partitions while horizontal boundaries separate private and shared segments. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

2. Optimization techniques

Getting good application performance on multi-core systems requires the collaboration between the application code and the underlying system software. Traditional sequential program optimizations, such as vectorization, data padding, prefetching and loop optimizations,[1] are all applicable to UPC programs since they improve single-core performance. In this paper, we focus on UPC-specific optimization techniques for multi-core systems.

For each optimization technique, we demonstrate its usage and show performance potentials using micro-benchmarks. Two representative multi-core systems, one with AMD processors and the other with Intel processors, are used for the experiments. The configurations of the two systems are listed in Table 1. The UPC compilers used in the experiments are Berkeley UPC 2.12 [15] and GCC UPC 4.5.1.2 [5], both of which are open source and free to use.

2.1. Casting shared pointers to local pointers

The global address space in UPC is partitioned across all threads and each shared datum in the global address space has unique affinity to one thread. In UPC, a shared pointer is a data structure that points to a shared datum in the global address space and a local pointer is a virtual memory address to which can be accessed by CPU directly. To access data through a shared pointer, the UPC compiler and runtime need to translate the shared pointer to a local pointer, an operation called address translation. However, if the memory location referred by a shared pointer has affinity with the accessing thread, the thread can access that memory location directly with a local pointer casted from the shared pointer and save the overheads for subsequent accesses. In the rest of the paper, the term "shared pointer" is the same as "pointer-to-shared" and "local pointer" is the same as "pointer-to-local".

A shared pointer in UPC commonly contains more information than a local pointer. A shared pointer typically includes the node rank, the address or variable name and the phase within an array block for the target data. Because the address translation form a shared pointer to a local pointer is usually done by software, accessing data through a shared pointer costs many more CPU cycles than through a local pointer. Casting a pointer-to-shared to a pointer-to-local on shared-memory systems whenever possible can avoid the address translation overheads and result in significant speed-ups.

We use a modified UPC STREAM TRIAD benchmark as an example to show the benefits of casting shared pointers to local pointers. Figures 2 and 3 are two versions of the same STREAM benchmark except that one uses shared pointers directly and the other uses local pointers by casting the shared pointers.

Figure 4 shows the performance difference between these two UPC programs. The program using local pointers is many times faster than the program using shared pointers. Not only does address translation incurs significant CPU overheads, it also wastes scarce memory bandwidth and interferes common hardware optimizations for local memory accesses such as prefetching. Though the problem has been an important research topic, the compiler generally cannot determine whether it is safe to cast a shared pointer to a local pointer due to the lack of runtime information. Therefore, it is best to cast shared pointers to local pointers explicitly and appropriately in the program to guarantee good performance.

2.2. Selecting memory consistency model

UPC supports two memory consistency models: *strict* and *relaxed*, which resemble the sequential consistency model [9] and the relaxed consistency model respectively. The *strict* consistency model provides a total ordering for all memory accesses and therefore it is easier to reason about. However, the *strict* consistency model also carries a huge performance penalty compared to the relaxed consistency model because it prohibits many kinds of valid concurrent data accesses. The details of these two UPC memory consistency models can be found in the UPC specification [16].

[1] http://en.wikipedia.org/wiki/Loop_optimization.

Table 1
Two multi-core NUMA systems

System	Sun Fire x4600-M2	IBM iDataPlex
Processor	AMD Opteron 8387	Intel Xeon E5530
Clock (GHz)	2.80	2.4
Cores per socket	4	4
Sockets	8	2
Total cores	32	8
Private L1 data cache	128 kB	64 kB
Private L2 data cache	512 kB	512 kB
Shared L3 cache per socket	6 MB	8 MB
Memory bandwidth (GB/s)	12.8	25.6
Memory	256 GB DDR2-667 ECC	24 GB DDR3-1066 ECC
Compiler	GCC 4.4.2	Intel C/C++ 11.1
Math library	ACML 4.3	Intel MKL 10.2

```
shared [] double *sa, *sb, *sc;
for (i=0; i<nelems; i++) {
  sa[i] = sb[i] + alpha * sc[i];
}
```

Fig. 2. Kernel code of the STREAM benchmark using shared pointers.

```
shared [] double *sa, *sb, *sc;
double *a, *b, *c;
a=(double *)sa;
b=(double *)sb;
c=(double *)sc;
for (i=0; i<nelems; i++) {
  a[i] = b[i] + alpha * c[i];
}
```

Fig. 3. Kernel code of the STREAM benchmark using local pointers.

The memory consistency model can be selected either by compiler directives:

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

or by type qualifiers:

```
strict shared [] double *sa;
relaxed shared [] double *sa.
```

The code used for benchmarking UPC memory consistency models is similar to the one in Fig. 2 used in Section 2.1 except for the memory consistency model. Figure 5 shows that using the relaxed consistency model can provide orders of magnitude faster performance than the strict consistency model on multi-

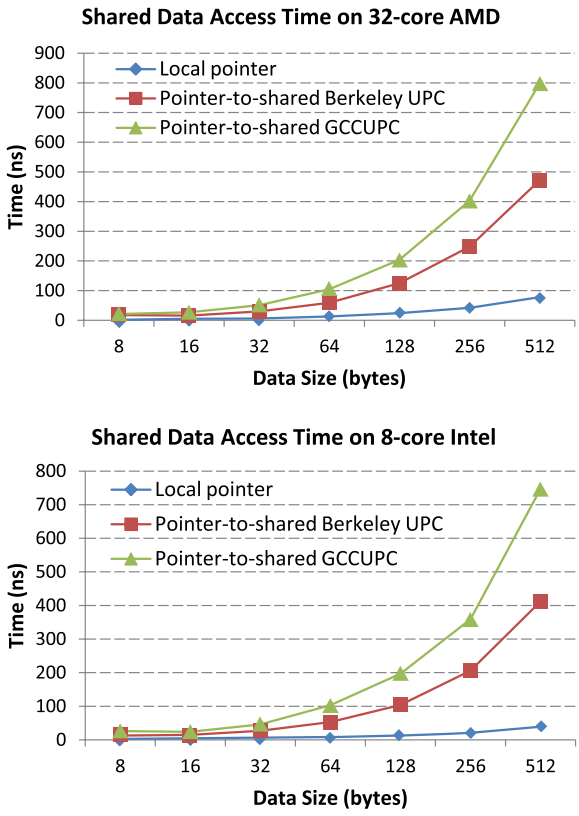

Fig. 4. Local shared data access times by pointer-to-local versus pointer-to-shared. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

core systems. This is because enforcing strict memory consistency requires expensive hardware instructions and sometimes even needs software assistance on some platforms. Using the relaxed consistency model

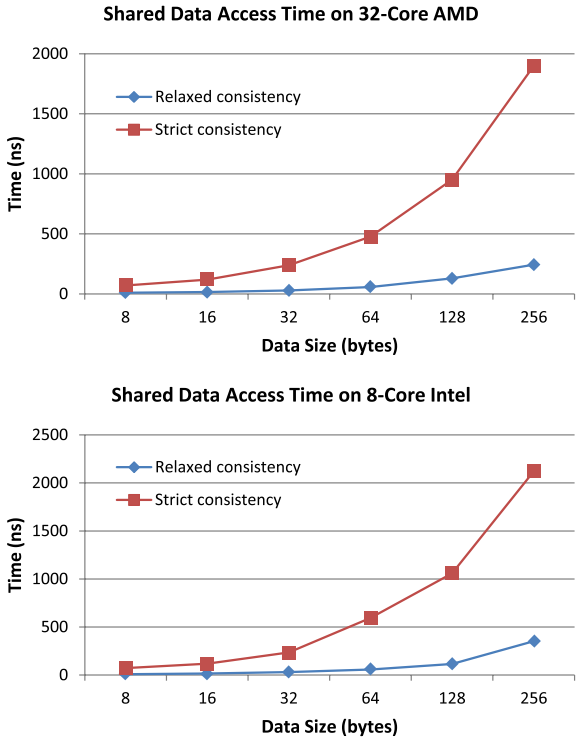

Fig. 5. Comparison of shared data access times between the relaxed consistency model and the strict consistency model. Berkeley UPC compiler is used for both systems. GCC is used on the AMD system and Intel C/C++ is used on the Intel system. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

whenever possible can greatly speed up the UPC program.

2.3. *Managing data affinity for NUMA systems*

Figure 6 shows the memory access latencies from thread 0 to different UPC threads as measured by the STREAM benchmark. Accessing the data with affinity to the thread itself has the lowest latency because of prefetching and caching. The memory access latencies are uniform within a socket and increase when the data are out of the socket where the accessing thread resides.

Elements in a UPC shared array are distributed in a block-cyclic order. Furthermore, UPC guarantees that the local portion of a shared array is contiguous in the thread's local memory. UPC programs can obtain the data affinity information by querying the UPC thread affinity function *upc_threadof* and the access distance information (available in Berkeley UPC extensions) to maximize local accesses and reduce remote accesses. It is important make the data access pattern match the data partition. A practical approach for getting good data locality is to run only one process on each socket (numa node) and spawn threads on the cores within the socket. Berkeley UPC supports this kind of process-thread hybrid execution model and includes runtime optimizations for multi-core systems [2].

2.4. *Using collective operations*

UPC provides commonly used collective operations in its standard library. Collective communication has been an important topic in parallel computing because it provides high level abstractions for describing communication patterns while allowing the underlying runtime and hardware to optimize the performance transparently. Although many MPI implementations include optimized collective communication, none of them supports thread-based collective communication because they use processes as communication endpoints. Multi-threaded collective operations for arrays are also absent in OpenMP.

Berkeley UPC provides optimized multi-threaded collective communication for multi-core systems [12] implemented with the GASNet communication library [4]. The GASNet multi-threaded collectives utilize automated-tuning to select the appropriate algorithm and adapt the system parameters to achieve best performance [10]. UPC applications can take advantage of these software engineering efforts in Berkeley UPC by simply calling UPC collective functions with threads.

Figure 7 shows the performance advantage of using the *broadcast* collective function on the 32-core AMD system. As the number of cores in shared-memory machines continues to increase, it will become more and more beneficial to use collective functions to express high level communication patterns.

The second example of using UPC collectives is to optimize matrix transpose. In this example, we also demonstrate the idea of using advanced data structures to store matrices in order to facilitate computation and communication.

We denote the N-by-N matrix by A, denote the transpose of A by A^{T} and denote the linear array for storing A by $\hat{A}$.

The direct transpose method by definition is:

$$A(i,j) = \hat{A}(i \times N + j), \tag{1}$$

$$A^{\mathrm{T}}(i,j) = A(j,i) = \hat{A}(j \times N + i). \tag{2}$$

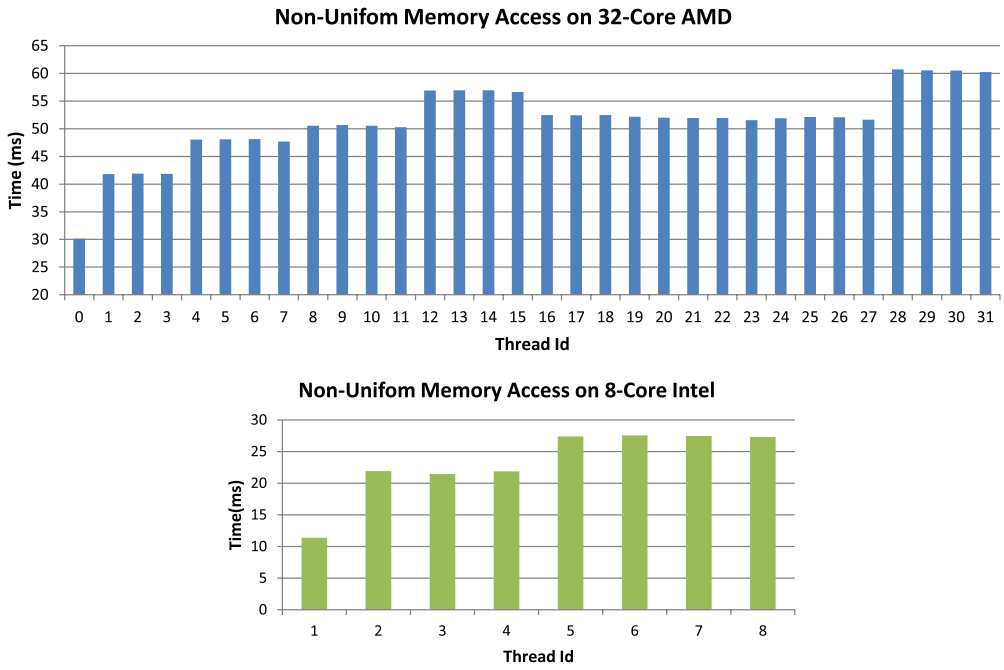

Fig. 6. Non-uniform memory access times measured by the latencies from thread 0 to data with affinity to other threads for streaming 32 MB of data. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

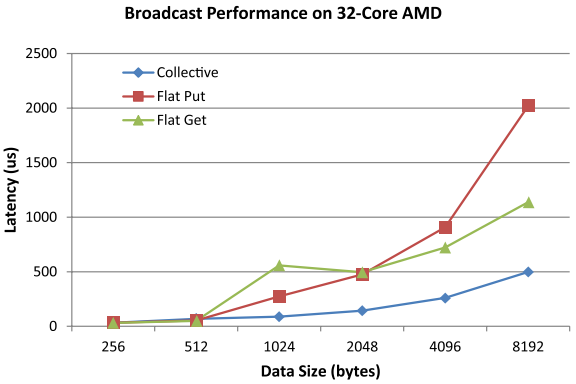

Fig. 7. Broadcast performance. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

Though the direct transpose method is simple to implement, it is very inefficient as it incurs many fine-grained non-contiguous data accesses by shared pointers. Instead, it is better to use a 2-D block-cyclic storage format with block size $B \times B$, which stores the elements in each block contiguously and distributes the blocks across all threads in a round-robin fashion. The optimized UPC version of transpose uses the *upc_all_exchange* collective operation to transpose the sub-matrices followed by efficient local transpose within each sub-matrix.

The index translation for 2-D block-cyclic distribution is given in Eqs 3 and 4:

$$A(i,j) = \hat{A}\big(i \times N + (i \bmod B) \times B^2 + j + (j \bmod B)\big), \tag{3}$$

$$A^{\mathrm{T}}(i,j) = A(j,i) = \hat{A}\big(j \times N + (j \bmod B) \times B^2 + i + (i \bmod B)\big). \tag{4}$$

This method can be implemented as follows:

```
B = N/THREADS;
nbytes = sizeof(double)*B*B;
upc_all_exchange(sb, sa, nbytes,
                 UPC_IN_MYSYNC | UPC_OUT_MYSYNC);

/* local transpose */
for (t=0; t<THREADS; t++) {
  la = (double *)&sa[MYTHREAD] + B*B*t;
  lb = (double *)&sb[MYTHREAD] + B*B*t;
  local_transpose(la, lb, B);
}
```

While the naive implementation of transpose has performance close to the collective version for very small matrix sizes, its run time increases dramatically as the problem size increases. As shown in Fig. 8, the

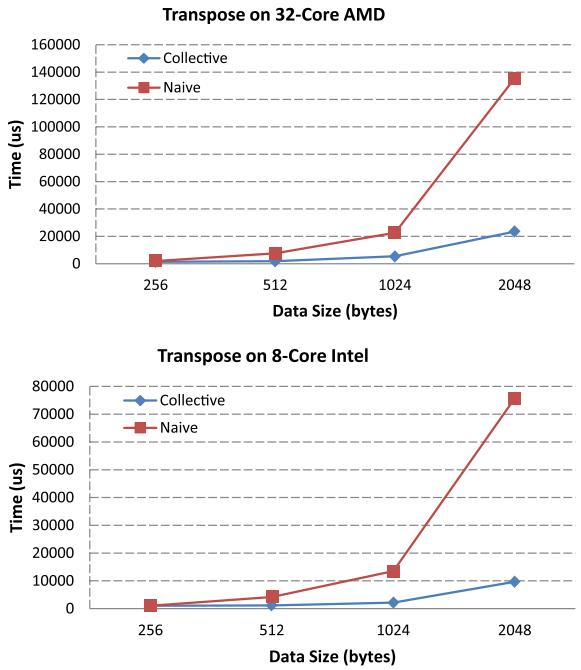

Fig. 8. Matrix transpose time. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

collective version of transpose is about 8 times faster than the naive version for a matrix of size 2048.

2.5. *Combining UPC with other programming models*

No single programming model or language can meet all users' needs and exploit diverse hardware architectures well. UPC programs can interact with application modules written in other programming languages and libraries. An example would be to use UPC for shared data management and reuse existing packages written in C, C++ and FORTRAN. In addition to sequential programming languages, UPC also supports interoperability with most current parallel programming models including MPI, OpenMP and CUDA. For the Berkeley UPC implementation, the translated code from the UPC source code is standard-compliant C code that can be compiled and linked by all major C compilers. Because the interoperability standard is still under development, there are some implementation-specific restrictions for mixing UPC with other programming models. To avoid deadlocks and data races, it is best to separate the use of different programming models in different phases bound by synchronization mechanisms such as barriers and memory fences.

3. Case studies

To provide a more complete picture of UPC program optimization, we use two important and easy to understand numerical computing kernels, dense matrix–matrix multiplication and 3-D Fast Fourier Transform, as case studies. Both of them are good examples for demonstrating UPC program optimizations that representative in many other computational problems. We co-design the data structure and the algorithm for the problems and apply the UPC optimization techniques discussed in the previous section to achieve good performance on multi-core systems.

3.1. *Dense matrix–matrix multiplication*

Dense matrix–matrix multiplication is an important numerical linear algebra kernel in the BLAS (Basic Linear Algebra Subprograms) library [3] that has been thoroughly studied and optimized. Most BLAS implementations use OpenMP or Pthreads to implement parallelism for shared-memory machines. The UPC implementation uses the vendor-optimized BLAS routines for single thread computation and use UPC to manage data distribution and communication across threads. The UPC version optimizes data locality and leverages collective operations.

The 2-D block-cyclic data distribution is key to the scalable performance because all processors can work in parallel on local data most of time. Each processor fetch remote data in large blocks to maximize memory bandwidth and hide memory latency.

We have applied the following optimizations to the UPC implementation of parallel matrix multiplication (Algorithm 1):

- Store the matrices in 2-D block-cyclic format as in Fig. 9 to maximize locality.
- Use optimized BLAS library for local dgemm.
- Overlap non-blocking one-sided communication with computation.
- Use team collective communication for row broadcast and column broadcast.

We compare the performance of the UPC dgemm implementation with the vendor-optimized BLAS libraries – ACML for the AMD system and MKL for the Intel system. As shown in Fig. 10, the UPC matrix–matrix multiplication routine not only outperforms the vendor-optimized libraries alone, but more importantly it is capable to run on distributed-memory machines without any change and therefore can save the developer significant amount of effort.

Algorithm 1 UPC matrix–matrix multiplication using the SUMMA algorithm

```
// Compute C = A × B, where A is M-by-P, B is P-by-N and C is M-by-N
// 2D processor grid TX × TY; block size: bs
// myrow: processor row id; mycol: processor column id;
for k = 0; k ⩽ P; k+ = bs do
  for i = 0; i ⩽ M; i+ = TX ∗ bs do
    Broadcast Block1 of size bs × bs starting at A(i+myrow*bs, k) to processors of the same row
    for j = 0; j ⩽ N; j+ = TY ∗ bs do
      Broadcast Block2 of size bs × bs starting at B(k, j+mycol*bs) to processors of the same column
      Compute local dgemm of Block1 and Block2 with vendor-optimized BLAS library
    end for
  end for
end for
```

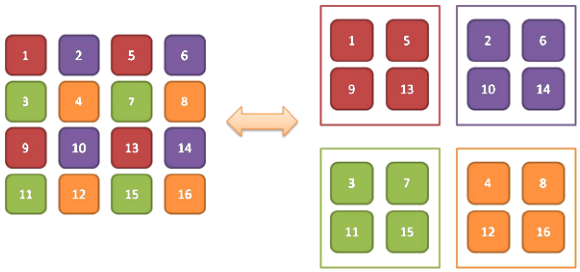

Fig. 9. Mapping between the global matrix view and the local data storage using 2-D block-cyclic data distribution for fast parallel matrix–matrix multiplication. This example assumes a processor grid of 2-by-2. The matrix blocks of the same color have data affinity to the same processor. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

3.2. 3-D FFT

Multi-dimensional Fast Fourier Transform (FFT) is another widely used numerical method for numerous applications. The most common multi-dimensional FFT algorithm is the row–column algorithm which performs 1-D FFT along each of the three dimensions successively. Because the 3-D array can only have one contiguous dimension and 1-D FFT is most efficient when the data are contiguous, it is necessary to transpose the data cube after each FFT phase to make the data to be used in the next FFT phase local and contiguous. This transpose step is implemented by the all-to-all collective communication. The key UPC optimization for the 3-D FFT problem is to choose a data distribution scheme that has good data locality and load balance.

We have applied the following optimization techniques to the UPC 3-D FFT implementation (Algorithm 2):

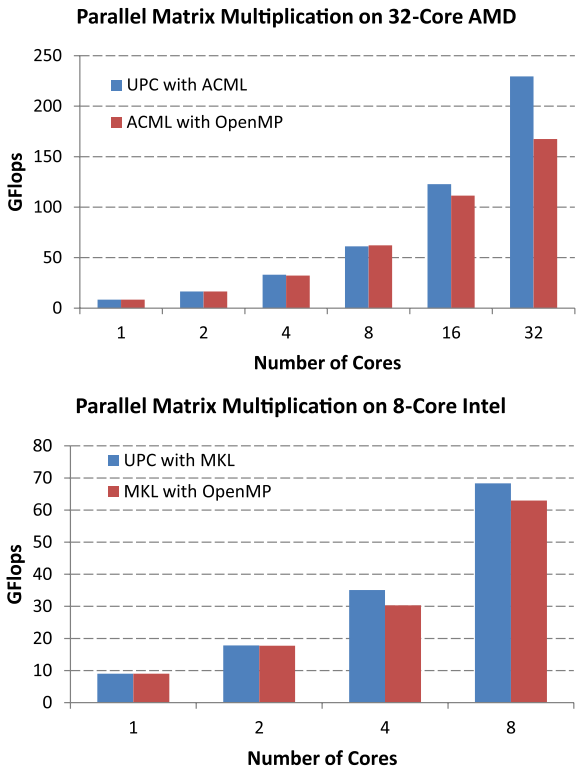

Fig. 10. Performance of matrix–matrix multiplication (dgemm) of two 8192-by-8192 real double floating-point matrices. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

- Partition the 3-D data array by a 2-D processor grid as in Fig. 11.
- Use optimized FFT library (FFTW) for local FFTs.
- Overlap one-sided communication with computation.
- Pin UPC threads to physical CPU cores to reduce migration overheads.

Algorithm 2 UPC 3D-FFT using the row–column algorithm

```
for all rows in the X dimension do
    Compute local 1-D FFT along the X dimension by FFTW
end for
Transpose the X-Y planes to make the array contiguous in the Y dimension
for all rows in the new Y dimension do
    Compute local 1-D FFT along the Y dimension by FFTW
end for
Transpose the Y-Z planes to make the array contiguous in the Z dimension
for all rows in the new Z dimension do
    Compute local 1-D FFT along the Z dimension by FFTW
end for
```

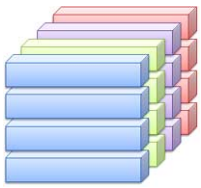

Fig. 11. 2-D decomposition for 3D-FFT with the row–column algorithm. The 3-D array is partitioned into slabs distributed evenly across all threads. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0310.)

Table 2
Performance of 3D-FFT (512 × 256 × 256) on the 32-core AMD system

Threads	4	8	16	32
FFTW (MFlops)	4561.3	7338.7	8756.4	8365.5
UPC with FFTW (MFlops)	2306.61	4242.28	7210.87	*9849.7*

- Aggregate small messages in the blocks.
- Use alltoall collective communication for array transpose.

As shown in Table 2, FFTW performs better than the UPC with FFTW implementation for small core counts because it has the opportunity to autotune for the whole multi-dimensional FFT problem. In contrast, the UPC with FFTW implementation only uses FFTW for local 1-D FFTs and FFTW just searches for the best 1-D FFT solution. However, the scalability of the UPC with FFTW implementation actually makes it has the best performance when running on 32 cores. The best 3-D FFT performance of 9849.7 MFlops on 32 cores is obtained by the UPC with FFTW implementation. In addition, this UPC FFT implementation can work on emerging non-cache-coherent architectures such as Intel Single-chip Cloud Computer and large scale clusters. The UPC 3-D FFT implementation is scalable on IBM BlueGene/P machines up to 32K cores [11].

4. Summary

UPC can help users get good performance on multi-core systems through expressing and managing application data locality with the PGAS model. Tools and libraries are key to UPC application developers' productivity. We list some common tools and libraries for UPC code development for further investigation by interested readers.

- Numerical library: UPC BLAS [6].
- Integrated Development Environment (IDE): IBM Parallel Tools Platform (PTP).[2]
- Debugger: GDB UPC,[3] Totalview.[4]
- Performance profiling and analysis: Parallel Performance Wizard[5] (PPW). The Berkeley UPC also provides basic tracing functionality which can be useful to locate communication hot spots.

Even though UPC was designed with the SPMD model by default, it is versatile enough for users to employ other programming styles such as fork-join and even dynamic threading [17] by user-level task management. For example, to emulate OpenMP's fork-join model, a UPC program may designate one thread as the master thread and the remaining threads as worker threads.

UPC supports both the MPI programming style and the shared-memory programming style. The MPI programming style would prefer explicit communication with bulk data transfers while the shared-memory programming style would use implicit fine-grained data transfers such as assignment statements. The under-

[2] http://www.eclipse.org/ptp/.
[3] http://www.gccupc.org/gdb-upc-info/gdb-upc-features.
[4] http://www.totalviewtech.com/.
[5] http://ppw.hcs.ufl.edu/.

lying hardware platforms impose constraints on how well UPC programs can perform when using different programming styles. The trade-off is that irregular applications are easier to be expressed in terms of fine-grained data accesses but most current interconnect hardware favors medium and large messages for transfers.

In addition, the Berkeley UPC implementation provides a number optimizations for multi-core systems:

- Multi-threaded collective communication for shared-memory machines [12].
- Fast shared-memory inter-process communication (IPC) [2].
- Thread and process scheduling and affinity management [7].

UPC is suitable for both large scale distributed-memory supercomputers as well as multi-core commodity systems and can be mixed with other programming models. It has an active user community with accessible tools and has demonstrated performance scalability, which together make UPC an attractive option for scientific and engineering application development.

References

[1] C. Barton, C. Caşcaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterjee and J.N. Amaral, Shared memory programming for large scale machines, in: *Programming Language Design and Implementation (PLDI)*, Ottawa, ON, Canada, 2006.

[2] F. Blagojevic, P. Hargrove, C. Iancu and K. Yelick, Hybrid PGAS runtime support for multicore nodes, in: *Partitioned Global Address Space (PGAS) Programming Models Conference*, 2010.

[3] BLAS Home Page, available at: http://www.netlib.org/blas/.

[4] GASNet home page, available at: http://gasnet.cs.berkeley.edu/.

[5] GCCUPC website, available at: http://www.gccupc.org/.

[6] J. González-Domínguez, M. Martín, G. Taboada, J. Touriño, R. Doallo and A. Gómez, A parallel numerical library for UPC, in: *15th International Euro-Par Conference, Euro-Par'09. Europar 2009-Parallel Processing*, Delft, The Netherlands, 2009, Lecture Notes in Computer Science, Vol. 5704, pp. 630–641.

[7] C. Iancu, S. Hofmeyr, F. Blagojevic and Y. Zheng, Oversubscription on multicore processors, in: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[8] S. Kamil, C. Chan, L. Oliker, J. Shalf and S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[9] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Transactions on Computers* **C-28**(9) (1979), 690–691.

[10] R. Nishtala, Automatically tuning collective communication for one-sided programming models, PhD thesis, University of California, Berkeley, CA, USA, 2009.

[11] R. Nishtala, P.H. Hargrove, D.O. Bonachea and K.A. Yelick, Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap, in: *23rd International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[12] R. Nishtala and K.A. Yelick, Optimizing collective communication on multicores, in: *HotPar'09: Proceedings of the Workshop on Hot Topics in Parallelism*, USENIX, March 2009.

[13] OPEN UH website, available at: http://www2.cs.uh.edu/~openuh.

[14] ROSE website, available at: http://www.rosecompiler.org.

[15] The Berkeley UPC Compiler, available at: http://upc.lbl.gov.

[16] UPC language specifications, v1.2, Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[17] B. Verastegui (ed.), *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007*, Reno, Nevada, USA, November 10–16, 2007, ACM Press, 2007.

[18] S. Williams, J. Carter, L. Oliker, J. Shalf and K. Yelick, Lattice Boltzmann simulation optimization on leading multicore platforms, in: *IEEE International Symposium on Parallel and Distributed Processing, 2008, IPDPS 2008*, April 2008, pp. 1–14.