Hindawi
Scientific Programming
Volume 2019, Article ID 6825728, 20 pages
https://doi.org/10.1155/2019/6825728

Research Article

Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC

Jérémie Lagravière,[1] Johannes Langguth,[1] Martina Prugger,[2] Lukas Einkemmer,[2] Phuong Hoai Ha,[3] and Xing Cai[1,4]

[1]*Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway*
[2]*University of Innsbruck, Technikerstraße 13, A-6020 Innsbruck, Austria*
[3]*The Arctic University of Norway, NO-9037 Tromsø, Norway*
[4]*University of Oslo, NO-0316 Oslo, Norway*

Correspondence should be addressed to Xing Cai; xingca@simula.no

Received 26 September 2018; Revised 14 January 2019; Accepted 27 January 2019; Published 3 March 2019

Academic Editor: Manuel E. Acacio Sanchez

Copyright © 2019 Jérémie Lagravière et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Unified Parallel C (UPC) programming language offers parallelism via logically partitioned shared memory, which typically spans physically disjoint memory subsystems. One convenient feature of UPC is its ability to automatically execute between-thread data movement, such that the entire content of a shared data array appears to be freely accessible by all the threads. The programmer friendliness, however, can come at the cost of substantial performance penalties. This is especially true when indirectly indexing the elements of a shared array, for which the induced between-thread data communication can be irregular and have a fine-grained pattern. In this paper, we study performance enhancement strategies specifically targeting such fine-grained irregular communication in UPC. Starting from explicit thread privatization, continuing with block-wise communication, and arriving at message condensing and consolidation, we obtained considerable performance improvement of UPC programs that originally require fine-grained irregular communication. Besides the performance enhancement strategies, the main contribution of the present paper is to propose performance models for the different scenarios, in the form of quantifiable formulas that hinge on the actual volumes of various data movements plus a small number of easily obtainable hardware characteristic parameters. These performance models help to verify the enhancements obtained, while also providing insightful predictions of similar parallel implementations, not limited to UPC, that also involve between-thread or between-process irregular communication. As a further validation, we also apply our performance modeling methodology and hardware characteristic parameters to an existing UPC code for solving a 2D heat equation on a uniform mesh.

1. Motivation

Good programmer productivity and high computational performance are usually two conflicting goals in the context of developing parallel code for scientific computations. *Partitioned global address space* (PGAS) [1–4], however, is a parallel programming model that aims to achieve both goals at the same time. The fundamental mechanism of PGAS is a global address space that is conceptually shared among concurrent processes that jointly execute a parallel program. Data exchange between the processes is carried out by a low-level network layer "under the hood" without explicit involvement from the programmer, thus providing good productivity. The shared global address space is logically partitioned such that each partition has affinity to a designated owner process. This awareness of data locality is essential for achieving good performance of parallel programs written in the PGAS model because the globally shared address space may actually encompass many physically distributed memory subsystems.

Unified Parallel C (UPC) [5, 6] is an extension of the C language and provides the PGAS parallel programming model. The concurrent execution processes of UPC are termed as *threads*, which execute a UPC program in the style of single-program-multiple-data. The data variables of each thread are of two types: *private* and *shared*. Variables of the

second type, accessible by all the threads, are found in the globally shared address space. In particular, shared data arrays of UPC provide programmer friendliness because any thread can use a global index to access an arbitrary array element. If the accessing thread does not own the target element, between-thread communication will be carried out automatically.

Another parallelization-simplifying feature of UPC is that shared arrays allow a straightforward distribution of data ownership among the threads. However, the simple data distribution scheme adopted for shared arrays may bring disadvantages. First, balancing the computational work among the threads can be more challenging than other parallel programming models that allow an uneven (static or dynamic) distribution of array elements to account for the possibly inhomogeneous cost per element. Second, for a UPC program where between-thread memory operations are inevitable, which is true for most scientific applications, the only mechanism for a programmer to indirectly control the impact of remote-memory traffic is to tune the block size constant of shared arrays. Third, all nonprivate memory operations (i.e., between threads) are considered in UPC to be of one type. There is no way for UPC to distinguish intra-compute-node memory operations (between threads running on the same hardware node) from their internode counterparts. The latter require explicitly transferring data over some interconnect between the nodes, which is considerably more costly.

In this paper, we will closely investigate the second and third disadvantages mentioned above. This will be done in the context of fine-grained remote-memory operations that have irregular thread-to-thread communication patterns. They can arise from irregular and indirectly indexed accesses to the elements of shared arrays in UPC. Our objectives are two-fold. First, we want to study the impact of several performance-enhancing techniques in UPC programming: privatizing for-loop iterations among threads, explicitly casting pointers-to-shared to pointers-to-local, adopting bulk memory transfers instead of individual remote-memory accesses, and message condensing with consolidation. Second, and more importantly, we will propose performance models for a representative example of scientific computations that induce fine-grained and irregular UPC remote-memory operations. Based on a simple philosophy of quantifying the occurrences and volumes of two categories of interthread memory traffic: local interthread traffic (within a compute node) and remote interthread traffic (between nodes), these performance models only need a small number of hardware characteristic parameters to provide a realistic performance prediction. This helps to understand and further tune the obtainable performance on an existing hardware system, while giving insightful predictions of the achievable scalability on upcoming new platforms.

We will focus on a specific category of matrix-vector multiplication operations where the involved matrix is sparse and has a constant number of nonzero values per row. Such a computational kernel appears in many branches of computational science. A performance challenge is that the nonzero values per matrix row can spread irregularly with respect to the columns. Consequently, any computer implementation will involve irregular and indirectly indexed accesses to numerical arrays. The resulting fine-grained irregular data accesses need to be handled with care, particularly in parallel implementations. We thus choose this computational kernel as a concrete case of fine-grained irregular communication that can occur in the UPC code. We want to demonstrate that proper code transformations can be applied to a naive, programmer-friendly but inefficient UPC implementation, for obtaining considerable enhancements of the computing speed. Moreover, the obtained performance enhancements can be backed up by conceptually simple performance models.

The remainder of this paper is organized as follows. Section 2 explains the basic mechanism of shared arrays, the cornerstone of UPC programming. Then, Section 3 gives a numerical description of our target computational problem and a naive UPC implementation. Thereafter, Section 4 shows in detail three programming strategies that transform the naive implementation for increasingly better performance. In Section 5, three performance models are developed to match with the three code transformations. Section 6 presents an extensive set of numerical experiments and time measurements both for showing the impact of the code transformations and verifying the performance models developed. Relevant related work is reviewed in Section 7, whereas Section 8 shows how our performance modeling methodology can easily be extended to simpler 2D calculations on a uniform mesh, before Section 9 concludes the entire paper with additional comments.

2. Shared Arrays of UPC

Shared data arrays, whose elements are freely accessible by all the threads, typically constitute the main data structure of a UPC program. They thus deserve a separate introduction in this section. The most common scenario is that the elements of a shared array have an evenly distributed thread affinity. This gives a straightforward approach to data partitioning while providing a user-controllable mechanism of data locality. The standard upc_all_alloc function (see e.g. [6]), to be used by all the UPC implementations in this paper for array allocation, has the following syntax:

shared void *upc_all_alloc (size_t nblks, size_t nbytes).

Note that shared is a specific type qualifier. Also, upc_all_alloc needs to be *collectively* called by all threads to allocate a shared array. Upon return, a private pointer to the allocated shared array, or a private *pointer-to-shared* in the more rigorous UPC terminology, becomes available on each thread. The allocated ssshared array consists of nblks blocks in total, whose affinity is distributed evenly among the threads in a *cyclic* order. The value of nbytes is the number of bytes occupied per block, which translates the block size, i.e., number of elements per block, as nbytes/sizeof(one element). The blocks that have affinity to the same owner thread are physically allocated *contiguously* in the owner thread's local memory.

This data ownership distribution scheme, which in many cases determines an associated work partitioning, has the advantage of a simple mapping between a global element index and the owner thread ID. It can be described as follows:

$$\text{owner_thread_id} = \left\lfloor \frac{\text{global_index}}{\text{block_size}} \right\rfloor \text{modulo THREADS}, \tag{1}$$

where THREADS is a built-in variable of UPC that stores the total number of threads participating in a parallel execution. The value of THREADS is fixed at either compile time or run time.

Accessing the elements of a shared array by their global indices, although programmer-friendly, can potentially incur considerable overhead. This is because a pointer-to-shared has three fields: the owner thread ID, the phase (i.e., the element offset within the affinity block), and the corresponding local memory address [5]. The standard upc_threadof(shared void *ptr) function of UPC returns the owner thread ID of an element that is pointed to by the pointer-to-shared ptr. Every access through a pointer-to-shared requires updating the three fields and thus always incurs overhead. Moreover, if the accessing thread is different from the owner thread, a behind-the-scene data transfer between the two threads has to be carried out. For indirectly indexed accesses of the elements in a shared array, a compiler cannot batch the individual between-thread data exchanges for the purpose of message aggregation (such as described in [7, 8]). The individual between-thread data exchanges are thus particularly costly when the accessing thread runs on a different compute node than the owner thread.

3. SpMV and a Naive UPC Implementation

This section is devoted to explaining the target computational kernel of this paper and presenting a naive UPC implementation.

3.1. Definition of Sparse Matrix-Vector Multiplication. Mathematically, a general matrix-vector multiplication is compactly denoted by $y = Mx$. Without loss of generality, we assume that the matrix M is square, having n rows and n columns. The input vector x and the result vector y are both of length n. Then, the general formula for computing element number i of the result vector y is as follows (using zero-based indices):

$$y(i) = \sum_{0 \le j < n} M(i, j)x(j). \tag{2}$$

If most of the $M(i, j)$ values are zero, M is called a *sparse* matrix. In this case, the above formula becomes unnecessarily expensive from a computational point of view. A more economic formula for computing $y(i)$ in a sparse matrix-vector multiplication (SpMV) is thus

$$y(i) = \sum_{M(i,j) \neq 0} M(i, j)x(j), \tag{3}$$

which only involves the nonzero values of matrix M on each row. Moreover, it is memory-wise unnecessarily expensive to store all the n^2 values of a sparse matrix, because only the nonzero values are used. This prompts the adoption of various compact storage formats for sparse matrices, such as the coordinate format (COO), compressed sparse row format (CSR), compressed sparse column format (CSC), and the EllPack format [9].

In particular, for sparse matrices that have a fixed number of nonzero values per row, it is customary to use the EllPack storage format, which conceptually uses two 2D tables. Both tables are of the same size, having n rows and the number of columns equaling the fixed number of nonzeros per row. The first table stores all the nonzero values of the sparse matrix, whereas the second table stores the corresponding column indices of the nonzeros. Moreover, if we assume that all the values on the main diagonal of a sparse matrix M are nonzero, which is true for most scientific applications, it is beneficial to split M as

$$M = D + A, \tag{4}$$

where D is the main diagonal of M and A contains the off-diagonal part of M. Then, a modified EllPack storage format can employ a 1D array of length n to store the entire main diagonal D. There is no need to store the column indices of these nonzero diagonal values, because their column indices equal the row indices by definition. Suppose r_{nz} now denotes the fixed number of nonzero off-diagonal values per row. For storing the nonzero values in the off-diagonal part A, it is customary to use two 1D arrays both of length $n \cdot r_{\text{nz}}$ (instead of two $n \times r_{\text{nz}}$ 2D tables), one stores the nonzero off-diagonal values consecutively row by row, whereas the other stores the corresponding integer column indices.

Following such a modified EllPack storage format, a straightforward sequential C implementation of SpMV is shown in Listing 1, where the integer array J contains the column indices of all nonzero off-diagonal values.

The sparsity pattern of the M matrix, i.e., where its nonzeros are located, is described by the array J of column indices. The actual pattern is matrix-dependent and irregular in general, meaning that each $x(i)$ value is irregularly used multiple times in computing several values in the result vector y. This has an important bearing on the achievable performance of a typical computer implementation, because the actual sparsity pattern of the M matrix may affect the level of data reuse in the different caches of a computer's memory system. Additionally, for the case of parallel computing, some of the values in the x vector have to be shared between processes (or threads). The irregular data reuse in the x vector will thus imply an irregular data sharing pattern. The resulting communication overhead is determined by the number of process pairs that need to share some values of the x vector, as well as the amount of shared data between each pair. The impact of these issues on different UPC implementations of SpMV will be the main subject of study in this paper. In the following, we first present a naive UPC implementation, whereas code transformation strategies that aim to improve the performance will be discussed in Section 4.

```
for (int i = 0; i < n; i++) {
   double tmp = 0.0;
   for (int j = 0; j < r_nz; j++)
      tmp += A[i * r_nz + j] * x[J[i * r_nz + j]];
   y[i] = D[i] * x[i] + tmp;
}
```

Listing 1: A straightforward sequential implementation of SpMV using a modified EllPack storage format.

3.2. A Naive UPC Implementation. The user-friendliness of UPC allows for an equally compact and almost identical implementation of the SpMV computational kernel (starting from the line of upc_forall in Listing 2) as the straightforward C implementation in Listing 1. An immediate advantage is that parallelization is automatically enabled through using the upc_forall construct of UPC, which deterministically divides the iterations of a for-loop among the threads. The five involved data arrays, which are all allocated by upc_all_alloc as shared arrays, are evenly distributed among the UPC threads in a block-cyclic order. More specifically, the arrays x, y, and D adopt a programmer-prescribed integer value, BLOCKSIZE, as their block size associated with the affinity distribution. The arrays A and J, both of length $n \cdot r_{nz}$, use $r_{nz} *$ BLOCKSIZE as their block size. This gives a consistent thread-wise data distribution for the five shared data arrays.

The UPC implementation of SpMV shown in Listing 2 is clean and easy to code. The parallelization details, i.e., data distribution and work partitioning, are an inherent part of the language definition of UPC. Since the number of nonzeros per matrix row is assumed to be fixed, the adopted thread-wise data distribution for the shared D, A, J, and y arrays is perfect, in that each thread will only access its owned blocks of these arrays. For the shared array x, whose values are indirectly accessed via the column index array J, underlying data transfers between the threads are inevitable in general.

As will be detailed later, the irregular column positions of the nonzero values (stored in the array J) will cause fine-grained and irregular between-thread data exchanges associated with the shared array x in the straightforward UPC implementation shown in Listing 2. Tuning the value of BLOCKSIZE can change the pattern and volume of between-thread communication. However, to ensure good performance, proper code transformations of such a naive UPC implementation are necessary.

4. Strategies of Performance Enhancement

This section studies three programming strategies that can be applied to transforming the naive UPC implementation. The main purpose is to reduce the impact of implicit between-thread data exchanges that are caused by irregular and indirectly indexed accesses to the shared array x. At the same time, we also want to eliminate some of the other types of avoidable overhead associated with UPC programming.

4.1. Explicit Thread Privatization. Some of the programmer-friendly features of UPC are accompanied with performance penalties. Relating to the naive UPC implementation of SpMV in Listing 2, these concern automatically dividing the iterations of a for-loop among threads by upc_forall and allowing any thread to access any element of a shared array. See Section 2 about the latter.

The upc_forall construct of UPC is a collective operation. In the example of upc_forall ($i = 0$; $i < n$; i++; &$y[i]$) used in Listing 2, all the threads go through the *entire* for-loop and check the affinity of each iteration, by comparing whether upc_threadof(&$y[i]$) equals the built-in MYTHREAD value that is unique per thread. Although only iterations having an affinity equaling MYTHREAD are executed by a thread, it is not difficult to see the overhead due to excessive looping and calls to the standard upc_threadof function behind the scene.

To get rid of the unnecessary overhead associated with upc_forall, we can let each thread work directly with the loop iterations that have the matching affinity. Note that the affinity distribution of the i-indexed loop iterations can be easily determined using the value of BLOCKSIZE. Such an explicit thread privatization of the loop iterations also opens up another opportunity for performance enhancement. Namely, all the globally indexed accesses to the shared arrays y, A, J, and D (except x) can be replaced by more efficient accesses through private pointers and local indices. This is achievable by using the well-known technique of casting pointers-to-shared to pointers-to-local [10]. Following these two steps that can be loosely characterized as *explicit thread privatization*, the naive UPC implementation can be transformed as shown in Listing 3.

It can be observed in Listing 3 that each thread now only traverses its designated rows of the sparse matrix. The computational work per thread is executed by going through its owned blocks in the shared arrays y, D, A, and J, for which each thread is guaranteed to never touch blocks owned by the other threads. Pointers to these four shared arrays are cast to their local counterparts loc_y, loc_D, loc_A, and loc_J, since array accesses through private pointers and local indices are the most efficient. On the other hand, casting pointer-to-shared x cannot be done, because the indirect accesses of form $x[\text{loc_}J[k * r_{nz} + j]]$ may lead to situations where the accessing thread is different from the owner thread. Compared with the naive UPC implementation in Listing 2, the transformed version after explicit thread privatization will have a much better performance. However, further performance improvement can be obtained by also "privatizing" the global accesses to the shared array x. This can be achieved by two code transformations, of different programming complexities and performance gains, which are to be detailed below.

4.2. Block-Wise Data Transfer between Threads. Although Listing 3 improves the naive UPC implementation by the approaches of explicit thread privatization, each thread still indirectly accesses the elements of the shared array x through global indices that are stored in the array J (now cast to pointer-to-local loc_J per block). When an indirectly

```
/* Total number of blocks in every shared array */
int nblks = n/BLOCKSIZE + (n% BLOCKSIZE) ?1 : 0;
/* Allocation of five shared arrays */
shared [BLOCKSIZE] double *x = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [BLOCKSIZE] double *y = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [BLOCKSIZE] double *D = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [r_nz * BLOCKSIZE] double *A = upc_all_alloc (nblks, r_nz * BLOCKSIZE * sizeof(double));
shared [r_nz * BLOCKSIZE] int *J = upc_all_alloc (nblks, r_nz * BLOCKSIZE * sizeof(int));
// ...
/* Computation of SpMV involving all threads */
upc_forall (int i = 0; i < n; i++; &y[i]) {
   double tmp = 0.0;
   for (int j = 0; j < r_nz; j++)
      tmp += A[i * r_nz + j] * x[J[i * r_nz + j]];
   y[i] = D[i] * x[i] + tmp;
}
```

Listing 2: A naive UPC implementation of SpMV using a modified EllPack storage format.

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Instead of upc_forall, each thread directly handles its designated blocks */
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (int mb = 0; mb < mythread_nblks; mb++) {
   int offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
   /* casting shared pointers to local pointers */
   double *loc_y = (double*) (y + offset);
   double *loc_D = (double*) (D + offset);
   double *loc_A = (double*) (A + offset * r_nz);
   int *loc_J = (int*) (J + offset * r_nz);
   /* computation per block */
   for (int k = 0; k < min(BLOCKSIZE, n-offset); k++) {
      double tmp = 0.0;
      for (int j = 0; j < r_nz; j++)
         tmp += loc_A[k * r_nz + j] * x[loc_J[k * r_nz + j]];
      loc_y[k] = loc_D[k] * x[offset + k] + tmp;
   }
}
```

Listing 3: An improved UPC implementation of SpMV by explicit thread privatization.

indexed $x[\text{loc_}J[k * r_{nz} + j]]$ value has affinity with MYTHREAD, the overhead only concerns updating the three fields of a pointer-to-shared. If MYTHREAD is different from the owner thread, however, a behind-the-scene data transfer will be executed in addition. Moreover, these between-thread data transfers will happen one by one, because a typical compiler is unable to batch the individual transfers. The extraoverhead is particularly high if the owner and accessing threads reside on different compute nodes. To avoid the potentially high overhead associated with $x[\text{loc_}J[k * r_{nz} + j]]$, we can create a private copy of x on each thread and transfer the needed blocks from x to the private copy before carrying out the SpMV. The resulting UPC implementation is shown in Listing 4.

In Listing 4, we have used the one-sided communication function upc_memget of UPC to transfer all the needed blocks, one by one, from the shared array x into a thread-private local copy named mythread_x_copy. The syntax of upc_memget is as follows:

> void upc_memget(void *dst, shared const void *src, size_t n).

We have thus completely avoided accessing values of the shared array x. However, there are a few "prices" paid on the way. First, each thread needs to allocate its own mythread_x_copy array of length n. This obviously increases the total memory usage. Second, the actual computation of SpMV on each thread needs to be preceded by transporting all the needed blocks from the shared array x into the corresponding places of mythread_x_copy. Specifically, a needed block from x is defined as having at least one $x[\text{loc_}J[k * r_{nz} + j]]$ value that will participate in calculating

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n * sizeof(double));
/* Prep-work: check for each block of x whether it has values needed by MYTHREAD; make a private boolean array
"block_is_needed" of length nblks */
// ...
/* Transport the needed blocks of x into mythread_x_copy */
for (int b = 0; b < nblks; b++)
  if (block_is_needed[b])
    upc_memget(&mythread_x_copy[b * BLOCKSIZE], &x[b * BLOCKSIZE], min(BLOCKSIZE, n − b * BLOCKSIZE) *
sizeof(double));
/* SpMV: each thread only goes through its designated blocks */
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (int mb = 0; mb < mythread_nblks; mb++) {
  int offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y + offset);
  double *loc_D = (double*) (D + offset);
  double *loc_A = (double*) (A + offset * r_nz);
  int *loc_J = (int*) (J + offset * r_nz);
  /* computation per block */
  for (int k = 0; k < min(BLOCKSIZE, n-offset); k++) {
    double tmp = 0.0;
    for (int j = 0; j < r_nz; j++)
      tmp += loc_A[k * r_nz + j] * mythread_x_copy[loc_J[k * r_nz + j]];
    loc_y[k] = loc_D[k] * mythread_x_copy[offset + k] + tmp;
  }
}
```

Listing 4: An improved UPC implementation of SpMV by block-wise communication.

the designated elements in y on each thread (with MYTHREAD as its unique ID). We note that each needed block is transported in its *entirety*, independent of the actual number of x values needed in that block. This also applies to the blocks of x that are owned by MYTHREAD. The whole procedure of transporting the needed blocks of x, implemented as the for-loop indexed by b in Listing 4, will result in time usage overhead. Nevertheless, this additional time usage is often compensated by avoiding the individual accesses to the shared array x. Third, to identify whether a block of x is needed by MYTHREAD requires prescreening the designated blocks of the array J (not shown in Listing 4). This is typically considered a negligible "one-time" cost if the same sparse matrix, or the same sparsity pattern shared among several sparse matrices, is repeatedly used in many SpMV operations later.

4.3. Message Condensing and Consolidation. One shortcoming of the transformed UPC code shown in Listing 4 is that each needed block from x is transported in its entirety. This will lead to unreasonably large messages, when only a small number of values in a block of x is needed by MYTHREAD. Also, several messages may be transported (instead of one consolidated message) between a pair of threads, where each message has a rigid length of BLOCKSIZE. To condense and consolidate the messages, we can carry out a different code transformation as follows.

4.3.1. Preparation Step. Each thread checks, in a "one-time" preparation step, which of its owned x values will be needed by the other threads. We also ensure that only one message is exchanged between each pair of communicating threads. The length of a message from thread T_1 to thread T_2 equals the number of *unique* values in the x blocks owned by T_1 that are needed by T_2. All the between-thread messages are thus condensed and consolidated. After this preparation step, the following private arrays are created on each thread:

int *mythread_num_send_values, *mythread_num_recv_values;

int **mythread_send_value_list, **mythread_recv_value_list;

double **mythread_send_buffers;

All the above private arrays have length THREADS (in the leading direction). If mythread_num_send_values[T] >0, it means that MYTHREAD needs to pack an outgoing message of this length for thread T as the receiver. Correspondingly, mythread_send_value_list[T] points to a list of local indices relative to a pointer-to-local, which is cast from &x[MYTHREAD * BLOCKSIZE], so that the respective needed x values can be efficiently extracted and packed together as the outgoing message mythread_send_buffers[T] toward thread T. The one-sided communication command upc_memput, which is of the following syntax:

```
void upc_memput(shared void *dst, const void *src,
size_t n)
```

will be used to transfer each outgoing message.

The meaning of mythread_num_recv_values[T] applies to the opposite communication direction. Also, the content of mythread_recv_value_list[T] will be needed by MYTHREAD to unpack the incoming message from thread T. One particular issue is that the upc_memput function requires a pointer-to-shared available on the destination thread. To this end, we need the following shared array with a block size of THREAD, where each array element is itself a pointer-to-shared:

```
shared[] double* shared [THREADS] shared_recv_buffers
[THREADS * THREADS];
```

An important task in the preparation step is to let each thread go through the following for-loop to allocate the individual buffers, in UPC's globally shared address space, for its expected incoming messages:

```
for (int T = 0; T < THREADS; T++)
  if (int length = mythread_num_recv_values[T] > 0)
    shared_recv_buffers[MYTHREAD * THREADS + T]
    = (shared[] double*)upc_alloc(length * sizeof
    (double));
```

It should be noted that the standard upc_alloc function should be called by only one thread. The entire array that is allocated by upc_alloc has affinity to the calling thread while being accessible by all the other threads [6]. In the above for-loop, each thread (using its unique MYTHREAD value) only calls upc_alloc inside its affinity block of shared_recv_buffers.

4.3.2. Communication Procedure. When the preparation step described above is done, we need to invoke a communication procedure to precede each SpMV computation. The communication procedure first lets each thread (with MYTHREAD as its unique ID) pack an outgoing message for every thread T that has mythread_num_send_values [T] > 0, by extracting the respective needed values from its owned blocks of the shared array x (cast to a pointer-to-local), using the local indices stored in mythread_send_value_list [T]. Then, the one-sided communication function upc_memput is called to send every ready-packed outgoing message to its destination thread. Thereafter, the upc_barrier command is posted to ensure that all the interthread communication is done, which means that all the expected messages have arrived on the respective destination threads. Finally, each thread unpacks every incoming message by copying its content to the respective positions in the thread-private array mythread_x_copy. Each thread also copies its owned blocks from the shared array x to the corresponding positions in the thread-privatemythread_x_copy. The entire communication procedure can be seen in Listing 5.

4.3.3. Implementation. By incorporating the above preparation step and communication procedure, we can create a new UPC implementation of SpMV in Listing 5. Specifically, each pair of communicating threads exchanges only one message containing the actually needed x values. As a "price," the new version has to introduce additional data structures in the preparation step and involve message packing and unpacking in the communication procedure.

5. Performance Models

We consider the performance model of a parallel implementation as a formula that can theoretically estimate the run time, based on some information of the target work and some characteristic parameters of the hardware platform intended. Roughly, the time usage of a parallel program that implements a computation comprises the time spent on the computational work and the parallelization overhead. The latter is mostly spent on various forms of communication between the executing processes or threads.

The three UPC implementations shown in Section 4 carry out identical computational work. However, they differ greatly in how the between-thread communication is realized, with respect to both the frequency and volume of the beween-thread data transfers. As will be demonstrated in Section 6, the time usages of the three transformed UPC implementations are very different. This motivates us to derive the corresponding performance models, with a special focus on modeling the communication cost in detail. Such theoretical performance models will help us to understand the actual computing speed achieved, while also providing hints on further performance tuning.

5.1. Time Spent on Computation. Due to a fixed number of nonzeros per matrix row, the amount of floating-point operations per thread is linearly proportional to the number of $y(i)$ values that are designated to each thread to compute. For all the UPC implementations in this paper, the shared array y is distributed in a block-cyclic manner, with a programmer-prescribed block size of BLOCKSIZE. Recall that the array y is of length n; thus, the number of y blocks assigned per thread, $B_{\text{thread}}^{\text{comp}}$, is given by the following formula:

$$B_{\text{total}}^{\text{comp}} = \left\lceil \frac{n}{\text{BLOCKSIZE}} \right\rceil,$$

$$B_{\text{thread}}^{\text{comp}} = \left\lfloor \frac{B_{\text{total}}^{\text{comp}}}{\text{THREADS}} \right\rfloor + \begin{cases} 1, & \text{if MYTHREAD} < \left(B_{\text{total}}^{\text{comp}} \text{ modulo THREADS}\right), \\ 0, & \text{else.} \end{cases} \tag{5}$$

Due to a low ratio between the number of floating-point operations and the induced amount of data movement in the memory hierarchy, the cost of computation for our SpMV example is determined by the latter, as suggested by the well-known Roofline model [11]. Our strategy is to derive the minimum amount of data movement needed between the main memory and the last-level cache. More specifically, the

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n * sizeof(double));
/* Preparation step: create and fill the thread-private arrays of int *mythread_num_send_values, int *mythread_num_recv_values,
int **mythread_send_value_list, int **mythread_recv_value_list, double **mythread_send_buffers. Also, shared_recv_buffers is
prepared. */
// ...
/* Communication procedure starts */
int T, k, mb, offset;
double *local_x_ptr = (double*)(x + MYTHREAD * BLOCKSIZE);
for (T = 0; T < THREADS; T++)
  if (mythread_num_send_values[T] > 0) /* pack outgoing messages */
    for (k = 0; k < mythread_num_send_values[T]; k++)
      mythread_send_buffers[T][k] = local_x_ptr[mythread_send_value_list[T][k]];
for (T = 0; T < THREADS; T++)
  if (mythread_num_send_values[T] > 0) /* send out messages */
    upc_memput(shared_recv_buffers[T * THREADS + MYTHREAD], mythread_send_buffers[T], mythread_num_send_values
    [T] * sizeof(double));
upc_barrier;
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (mb = 0; mb < mythread_nblks; mb++) /* copy own x-blocks */
  offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  memcpy(&mythread_x_copy[offset], (double*)(x + offset), min(BLOCKSIZE, n-offset) * sizeof(double));
}
for (T = 0; T < THREADS; T++)
  if (mythread_num_recv_values[T] > 0) {/* unpack incoming messages */
    double *local_buffer_ptr = (double*) shared_recv_buffers[MYTHREAD * THREADS + T];
    for (k = 0; k < mythread_num_recv_values[T]; k++)
      mythread_x_copy[mythread_recv_value_list[T][k]] = local_buffer_ptr[k];
  }
/* Communication procedure ends */
/* SpMV: each thread only goes through its designated blocks */
for (mb = 0; mb < mythread_nblks; mb++) {
  offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y + offset);
  double *loc_D = (double*) (D + offset);
  double *loc_A = (double*) (A + offset * r_nz);
  int *loc_J = (int*) (J + offset * r_nz);
  /* computation per block */
  for (k = 0; k < min(BLOCKSIZE, n-offset); k++) {
    double tmp = 0.0;
    for (int j = 0; j < r_nz; j++)
      tmp += loc_A[k * r_nz + j] * mythread_x_copy[loc_J[k * r_nz + j]];
    loc_y[k] = loc_D[k] * mythread_x_copy[offset + k] + tmp;
  }
}
```

Listing 5: An improved UPC implementation of SpMV by message condensing and consolidation.

following formula gives the minimum data traffic (in bytes) from/to the main memory for computing each $y(i)$ value:

$$D_{\min}^{\text{comp}} = r_{\text{nz}} \cdot (\text{sizeof}(\text{double}) + \text{sizeof}(\text{int})) + 3 \cdot \text{sizeof}(\text{double}), \tag{6}$$

where r_{nz} denotes the fixed number of off-diagonal nonzero values per matrix row, each occupying sizeof(double) bytes in memory, with sizeof(int) bytes needed per column index. The last term in (6) corresponds to the two memory loads for accessing loc_$D[k]$ and mythread_x_copy[offset + k] (or x[offset + k]) and the memory store associated with updating loc_$y[k]$. We refer to Listings 3–5 for the implementation details.

Formula (6) has assumed perfect data reuse in the last-level data cache. Our earlier experiences with the same SpMV computation (implemented in sequential C or OpenMP), for the case of a "proper" ordering of the matrix rows ([12]), suggest that (6) is a realistic estimate for the last two UPC implementations (Listings 4 and 5). For these two

implementations, the x values are fetched from the thread-private array mythread_x_copy. In the first transformed UPC implementation (Listing 3), indirectly indexed accesses to the shared array x (of form $x[\text{loc_}J[k * r_{\text{nz}} + j]]$) will incur additional memory traffic on "remote" threads, caused by the inevitable between-thread data transfers. We have chosen for this case to model the deviation from (6) as a part of the communication cost, to be discussed in Section 5.2.3.

Therefore, the minimum computational time needed per thread, which is the same for all the UPC implementations of this paper, can be estimated as

$$T_{\text{thread}}^{\text{comp}} = \frac{B_{\text{thread}}^{\text{comp}} \cdot \text{BLOCKSIZE} \cdot D_{\text{min}}^{\text{comp}}}{W_{\text{thread}}^{\text{private}}}, \tag{7}$$

where $W_{\text{thread}}^{\text{private}}$ denotes the realistic bandwidth (bytes per second) at which a thread can access its private memory space. This can be found by running a multithreaded STREAM benchmark [13] on one compute node of a target hardware platform, using the intended number of UPC threads per node. The $W_{\text{thread}}^{\text{private}}$ value equals the measured multithreaded STREAM bandwidth divided by the number of threads used. Note that the bandwidth measured by a single-threaded STREAM benchmark cannot be used directly as $W_{\text{thread}}^{\text{private}}$, unless a single UPC thread per compute node is indeed intended. This is because the multithreaded STREAM bandwidth is not linearly proportional to the number of threads used, due to saturation of the memory bandwidth.

5.2. Communication Overhead

5.2.1. Definitions. Before we dive into the details of modeling the various communication costs that are associated with the three transformed UPC implementations, it is important to establish the following definitions:

(i) If a thread accesses a memory location in the globally shared address space with affinity to another thread, a *non-private* memory operation is incurred.

(ii) A nonprivate memory operation, which is between two threads, can belong to one of two categories: *local inter-thread* and *remote inter-thread*. The first category refers to the case where the two involved threads reside on the same compute node, which has a physically shared NUMA (or UMA) memory encompassing all the threads running on the node. The second category refers to the case where the two threads reside on two different nodes, which need to use some interconnect for exchanging data.

(iii) A nonprivate memory operation, in each category, can happen in two modes: either individually or inside a sequence of memory operations accessing a contiguous segment of nonprivate memory. We term the first mode as *individual* and the second mode as *contiguous*.

5.2.2. Cost of Nonprivate Memory Operations. The time needed by one nonprivate memory operation, in the contiguous mode, can be estimated as

$$T_{\text{cntg}}^{\text{local}} = \frac{\text{sizeof (one element)}}{W_{\text{thread}}^{\text{local}}}, \qquad T_{\text{cntg}}^{\text{remote}} = \frac{\text{sizeof (one element)}}{W_{\text{node}}^{\text{remote}}}, \tag{8}$$

where $W_{\text{thread}}^{\text{local}}$ denotes the per-thread bandwidth for contiguous local interthread memory operations, and we assume for simplicity $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$, with the latter being defined in Section 5.1. Correspondingly, $W_{\text{node}}^{\text{remote}}$ denotes the interconnect bandwidth available to a node for contiguous remote (internode) memory operations. The reason for adopting a per-node bandwidth for internode memory operations is because the internode network bandwidth can typically be fully utilized by one thread, unlike the main-memory bandwidth. The value of $W_{\text{node}}^{\text{remote}}$ can be measured by a modified UPC STREAM benchmark or simply a standard MPI ping-pong test, to be discussed in Section 6.2.

The cost of one individual remote interthread memory operation, $T_{\text{indv}}^{\text{remote}}$, is assumed to be dominated by a constant latency overhead, denoted by τ. Specifically, the latency τ is independent of the actual number of bytes involved in one individual remote-memory operation. By the same reason, $W_{\text{node}}^{\text{remote}}$ has no bearing on $T_{\text{indv}}^{\text{remote}}$. The actual value of τ can be measured by a special UPC microbenchmark, to be discussed in Section 6.2. The cost of one individual local interthread memory operation can be estimated by the following formula:

$$T_{\text{indv}}^{\text{local}} = \frac{\text{sizeof (cache line)}}{W_{\text{thread}}^{\text{local}}}. \tag{9}$$

Here, we will again adopt $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$. The reason for having the size of one cache line as the numerator in (9) is that individual local interthread memory operations are considered to be noncontiguously spread in the private memory of the owner thread, thus paying the price of an entire cache line per access (it has been implied that one data element occupies fewer bytes than one cache line).

5.2.3. Communication Time for the First Transformed UPC Implementation. For the UPC implementation in Listing 3, an individual nonprivate memory operation arises when the owner thread of value $x[\text{loc_}J[k * r_{\text{nz}} + j]]$ is different from the accessing thread. Each such nonprivate memory operation costs either $T_{\text{indv}}^{\text{local}}$ as defined in (9) or $T_{\text{indv}}^{\text{remote}} = \tau$. To quantify the total communication time incurred per thread, we need the following two counts, which can be obtained by letting each thread examine its owned blocks of the shared array J:

(i) $C_{\text{thread}}^{\text{local,indv}}$: number of occurrences when $\&x[\text{loc_}J[k * r_{\text{nz}} + j]]$ has a different affinity than MYTHREAD and the owner thread resides on the same compute node as MYTHREAD

(ii) $C_{\text{thread}}^{\text{remote,indv}}$: number of occurrences when $\&x[\text{loc_}J[k * r_{\text{nz}} + j]]$ has a different affinity than MYTHREAD and the owner thread resides on a different compute node

Thus, the total communication cost per thread during each SpMV is

$$T_{\text{thread}}^{\text{comm,UPCv1}} = C_{\text{thread}}^{\text{local,indv}} \cdot \frac{\text{sizeof (cache line)}}{W_{\text{thread}}^{\text{private}}} + C_{\text{thread}}^{\text{remote,indv}} \cdot \tau. \tag{10}$$

5.2.4. Communication Time for the Second Transformed UPC Implementation. For the UPC implementation in Listing 4, before computing the SpMV, each thread calls the upc_memget function to transport its needed blocks from the shared array x to the private array mythread_x_copy. To estimate the communication time spent per node, we will use the following formula:

$$\begin{aligned} T_{\text{node}}^{\text{comm,UPCv2}} = & \max_{\forall \text{threads in node}} B_{\text{thread}}^{\text{local}} \\ & \cdot \frac{2 \cdot \text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}} \\ & + \sum_{\forall \text{threads in node}} B_{\text{thread}}^{\text{remote}} \\ & \cdot \left(\tau + \frac{\text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{node}}^{\text{remote}}} \right), \end{aligned} \tag{11}$$

where $B_{\text{thread}}^{\text{local}}$ denotes the number of x blocks residing on the same node as MYTHREAD and having at least one value needed by MYTHREAD, whereas $B_{\text{thread}}^{\text{remote}}$ denotes the number of needed blocks residing on other nodes. The reason for having a factor of 2 in the numerator of the first term on the right-hand side of (11) is due to the private/local memory loads and stores that both take place on the same node. Note that we have consistently assumed $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$. More importantly, we consider that all the threads on the same node concurrently carry out their intranode part of communication, whereas the internode operations of upc_memput are carried out one by one. For communicating each internode block, we have included τ as the "start-up" overhead in addition to the $W_{\text{node}}^{\text{remote}}$-determined cost.

5.2.5. Communication Time for the Third Transformed UPC Implementation. For the UPC implementation in Listing 5, the overhead per thread for preparing the private array mythread_x_copy before the SpMV has four parts: (1) packing all its outgoing messages, (2) calling upc_memput for each outgoing message, (3) copying its own blocks of x to the corresponding positions in mythread_x_copy, and (4) unpacking the incoming messages.

Let us denote by $S_{\text{thread}}^{\text{local,out}}$ the accumulated size of the outgoing messages from MYTHREAD to threads residing on the same node as MYTHREAD; $S_{\text{thread}}^{\text{remote,out}}$ denotes the accumulated size of the outgoing messages towards other nodes. Similarly, $S_{\text{thread}}^{\text{local,in}}$ and $S_{\text{thread}}^{\text{local,in}}$ denote the incoming counterparts. Then, the per-thread overhead of packing the outgoing messages is

$$T_{\text{thread}}^{\text{pack}} = \frac{\left(S_{\text{thread}}^{\text{local,out}} + S_{\text{thread}}^{\text{remote,out}}\right)(2 \cdot \text{sizeof (double)} + \text{sizeof (int)})}{W_{\text{thread}}^{\text{private}}}. \tag{12}$$

We remark that packing each value in an outgoing message requires loading at least sizeof (double)+ sizeof (int) bytes from the private memory and storing sizeof(double) bytes into the message.

Instead of modeling the per-thread overhead related to the upc_memput calls, we choose to model the per-node counterpart as

$$\begin{aligned} T_{\text{node}}^{\text{memput,UPCv3}} = & \max_{\forall \text{threads in node}} \frac{2 \cdot S_{\text{thread}}^{\text{local,out}} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}} \\ & + \sum_{\forall \text{threads in node}} \\ & \cdot \left(C_{\text{thread}}^{\text{remote,out}} \cdot \tau + \frac{S_{\text{thread}}^{\text{remote,out}} \cdot \text{sizeof (double)}}{W_{\text{node}}^{\text{remote}}} \right), \end{aligned} \tag{13}$$

where $C_{\text{thread}}^{\text{remote,out}}$ denotes the number of outgoing internode messages from MYTHREAD. Again, for each internode message, we have included τ as the "start-up" overhead in addition to the $W_{\text{node}}^{\text{remote}}$-determined cost.

The per-thread overhead of copying the private blocks of x into mythread_x_copy is

$$T_{\text{thread}}^{\text{copy}} = \frac{2 \cdot B_{\text{thread}}^{\text{comp}} \cdot \text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}}, \tag{14}$$

where we recall that $B_{\text{thread}}^{\text{comp}}$ is defined in (5).

Finally, the per-thread overhead of unpacking the incoming messages is

$$T_{\text{thread}}^{\text{unpack}} = \frac{\left(S_{\text{thread}}^{\text{local,in}} + S_{\text{thread}}^{\text{remote,in}}\right)(\text{sizeof (double)} + \text{sizeof (int)} + \text{sizeof (cache line)})}{W_{\text{thread}}^{\text{private}}}. \tag{15}$$

Note that sizeof(double) + sizeof(double)(int) corresponds to contiguously reading each value from an incoming message, whereas sizeof(cache line) corresponds to the cost of writing the value to a noncontiguous location in the array mythread_x_copy.

5.3. Total Time Usage. Due to the possible imbalance of both computational work and communication overhead among the threads, the total time usage of any of the UPC implementations will be determined by the slowest thread or node. For the first transformed UPC implementation, shown in Listing 3, the total time is determined by the slowest thread:

$$T_{\text{total}}^{\text{UPCv1}} = \max_{\forall \text{threads}} \left(T_{\text{thread}}^{\text{comp}} + T_{\text{thread}}^{\text{comm,UPCv1}} \right). \tag{16}$$

For the second transformed UPC implementation, shown in Listing 4, the total time is determined by the slowest node:

$$T_{\text{total}}^{\text{UPCv2}} = \max_{\forall \text{nodes}} \left(\left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{comp}} \right) + T_{\text{node}}^{\text{comm,UPCv2}} \right). \tag{17}$$

For the third transformed UPC implementation, shown in Listing 5, due to the needed explicit barrier after the upc_memput calls, the total time usage is modeled as

$$T_{\text{total}}^{\text{UPCv3}} = \max_{\forall \text{nodes}} \left(\left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{pack}} \right) + T_{\text{node}}^{\text{memput,UPCv3}} \right) + \max_{\forall \text{threads}} \left(T_{\text{thread}}^{\text{copy}} + T_{\text{thread}}^{\text{unpack}} + T_{\text{thread}}^{\text{comp}} \right). \tag{18}$$

5.4. Remarks. It is important to separate two types of information needed by the above performance models. The hardware-specific information includes $W_{\text{thread}}^{\text{private}}$, $W_{\text{node}}^{\text{remote}}$, τ, and the cache line size of the last-level cache. The first parameter denotes the per-thread rate of contiguously accessing private memory locations. The second parameter is the per-node counterpart for contiguously accessing remote off-node memory locations. Note that we do not distinguish between $W_{\text{thread}}^{\text{private}}$ and intrasocket or intersocket local memory bandwidths, due to very small differences between them. The τ parameter describes the latency for an individual remote-memory access. All the hardware parameters are easily measurable by simple benchmarks, see Section 6.2, or known from hardware specification.

The computation-specific information includes $C_{\text{thread}}^{\text{local,indv}}$, $C_{\text{thread}}^{\text{remote,indv}}$ (Section 5.2.3), $B_{\text{thread}}^{\text{local}}$, $B_{\text{thread}}^{\text{remote}}$ (Section 5.2.4), and $S_{\text{thread}}^{\text{local,out}}$, $S_{\text{thread}}^{\text{local,in}}$, $C_{\text{thread}}^{\text{remote,out}}$, $S_{\text{thread}}^{\text{remote,out}}$, and $S_{\text{thread}}^{\text{remote,in}}$ (Section 5.2.5). These numbers depend on the specific spread of the nonzero values in the sparse matrix. They can be obtained by letting each thread go through its owned blocks of the shared array J and do an appropriate counting. Another important input is the programmer-chosen value of BLOCKSIZE, which controls how all the shared arrays are distributed among the threads, thus determining all the above computation-specific parameters.

6. Experiments

To study the impact of various code transformations described in Section 4 and to validate the corresponding performance models proposed in Section 5, we will use a real-world case of SpMV in this section.

6.1. A 3D Diffusion Equation Solver Based on SpMV. One particular application of SpMV can be found in numerically solving a 3D diffusion equation that is posed on an irregular domain. Typically, an unstructured computational mesh must be used to match the irregular domain. All numerical strategies will involve a time integration process. During time step ℓ, the simplest numerical strategy takes the form of an SpMV:

$$v^{\ell} = M v^{\ell-1}, \tag{19}$$

where vectors v^{ℓ} and $v^{\ell-1}$ denote the numerical solutions on two consecutive time levels, each containing approximate values on some mesh entities (e.g., the centers of all tetrahedrons). The M matrix arises from a numerical discretization of the original diffusion equation. Matrix M is normally time-independent and thus computed once and for all, prior to the time integration process. The unstructured computational mesh will lead to an irregular spread of the nonzeros. Particularly, if a second-order finite volume discretization is applied to a tetrahedral mesh, the number of off-diagonal nonzero values per row of M is up to 16 [14].

Three test problems of increasing resolution will be used in this section. They all arise from modeling the left cardiac ventricle of a healthy male huma. (the 3D diffusion solver can be an integral part of a heart simulator). The three corresponding tetrahedral meshes are generated by the open-source *TetGen* software [15], with the actual size of the meshes being listed in Table 1. Note that we have $r_{\text{nz}} = 16$ for all the three test problems. The tetrahedrons have been reordered in each mesh for achieving good cache behavior associated with a straightforward sequential computation. It is important to notice that all the three meshes are fixed for the following UPC computations, independent of the number of UPC threads used and the value of BLOCKSIZE chosen.

For any computer program implementing the 3D diffusion solver, two arrays are sufficient for containing the two consecutive numerical solutions v^{ℓ} and $v^{\ell-1}$. For the UPC implementations discussed in Section 4, the shared array y corresponds to v^{ℓ} and x to $v^{\ell-1}$ during each time step. The pointers-to-shared y and x need to be swapped before the next time step, fenced between a pair of upc_barrier calls.

6.2. Hardware and Software Platforms. The Abel computer cluster [16] was used to run all the UPC codes and measure their time usage. Each compute node on Abel is equipped with two Intel Xeon E5-2670 2.6 GHz 8-core CPUs and 64 GB of RAM. The interconnect between the nodes is FDR InfiniBand (56 Gbits/s). With a multithreaded STREAM [13] microbenchmark in C, we measured the aggregate memory bandwidth per node as 75 GB/s using 16 threads. This gave

Table 1: Size of the three test problems.

	Test problem 1	Test problem 2	Test problem 3
Number of tetrahedrons, n	6,810,586	13,009,527	25,587,400

$W_{\text{thread}}^{\text{private}}$ = (75/16)GB/s. The internode communication bandwidth, $W_{\text{node}}^{\text{remote}}$ (defined in Section 5.2.2), was measured by a standard MPI ping-pong microbenchmark to be about 6 GB/s.

The Berkeley UPC [17] version 2.24.2 was used for compiling and running all our UPC implementations of SpMV. The compilation procedure involved first a behind-the-scene translation from UPC to C done remotely at Berkeley via HTTP, with the translated C code being then compiled locally on Abel using Intel's icc compiler version 15.0.1. The compilation options were -O3 -wd177 -wd279 -wd1572 -std = gnu99.

In order to measure the cost of an individual remote-memory transfer, τ (defined in Section 5.2.2), we developed a microbenchmark shown in Listing 6. Specifically, v is a shared UPC array created by upc_all_alloc. Each thread then randomly reads entries of v that have affinity with "remote threads," through a thread-private index array mythread_indices. The total time usage, subtracting the time needed to contiguously traverse mythread_indices, can be used to quantify τ. When using two nodes each running 8 UPC threads, we measured the value of τ as 3.4 μs. Varying the number of concurrent threads does not change the measured value of τ very much.

6.3. Time Measurements. Table 2 compares the performance of the naive UPC implementation (Listing 2) against that of the first transformed UPC implementation (Listing 3). Here, we only used one compute node on Abel while varying the number of UPC threads. Each experiment was repeated several times, and the best time measurement is shown in Table 2. Thread binding was always used, as for all the subsequent experiments. Test problem 1 (with 6810586 tetrahedrons) was chosen with the value of BLOCKSIZE being fixed at 65536. We can clearly see from Table 2 that the naive UPC implementation is very ineffective due to using upc_forall and accessing y, D, A, and J through pointers-to-shared.

Table 3 summarizes the time measurements for all the three transformed UPC implementations (denoted by UPCv1, 2, 3) and all the three test problems. It can be seen that UPCv3 has the best performance as expected, followed by UPCv2 with UPCv1 being the slowest. The only exception is that UPCv1 is faster than UPCv2 when running 16 UPC threads on one Abel node. This is because there is no "penalty" of individual remote-memory transfers for UPCv1 in such a scenario, whereas UPCv2 has to transfer all the needed blocks in entirety.

6.4. Validating the Performance Models. We have seen in Section 6.3 that the three transformed implementations have quite different performance behaviors. To shed some light on the causes of the performance differences, we will now use the hardware characteristic parameters obtained in Section 6.2 together with the performance models proposed in Section 5. Specifically, Table 4 compares the actual time measurements against the predicted time usages for Test problem 1 on the Abel cluster. It can be seen that the predictions made by the performance models of Section 5 follow the same trends of the actual time measurements, except for the case of UPCv1 using 128 threads. It is worth noticing that the single-node performance (16 threads) of UPCv2 is correctly predicted to be slower than that of UPCv1, whereas the reverse of the performance relationship when using multiple nodes is also confirmed by the predictions. For small thread counts (16–64), the prediction accuracy is quite good. For larger threads counts, the predictions become less accurate.

For UPCv1, there are four cases where the actual run times are faster than the predictions. These are attributed to the fact that the adopted τ value of 3.4 μs can be a little "pessimistic." Recall from Section 6.2 that the particular τ value was measured by the microbenchmark when it used 8 threads on one node to simultaneously communicate with 8 other threads on another node. In reality, the effective τ value can be smaller than 3.4 μs, if the average number of remotely communicating threads per node over time is fewer than 8. For UPCv3, there are two cases where the actual run times are slightly faster than the predictions. This is due to imbalance between the threads with respect to the per-thread amount of computation and message packing/unpacking. When most of the threads have finished their tasks, the remaining threads will each have access to an effective $W_{\text{thread}}^{\text{private}}$ value that is larger than 1/16 of $W_{\text{node}}^{\text{private}}$. This can result in the time prediction of UPCv3 being a little "pessimistic."

To examine some of the prediction details of UPCv3, we show in Figure 1 the per-thread measurements and predictions of $T_{\text{thread}}^{\text{comp}}$, $T_{\text{thread}}^{\text{unpack}}$, and $T_{\text{thread}}^{\text{pack}}$ (Section 5.2.5), for the particular case of using 32 threads on two nodes. It can be seen that the predictions of the three time components closely match the actual time usages.

As mentioned in Section 4, the three UPC implementations differ in how the interthread communications are handled. To clearly show the difference, we have plotted in the top of Figure 2 the per-thread distribution of communication volumes for the specific case of using 32 threads with BLOCKSIZE set to 65536. We observe that UPCv3 has the lowest communication volume, whereas UPCv2 has the highest. Although UPCv1 induces lower communication volumes than UPCv2, all communications of the former are individual and thus more costly. It is also observed that the communication volumes can vary considerably from thread to thread. The specific variation depends on the spread of the nonzeros, as well as the number of threads used and the value of BLOCKSIZE chosen. The dependency on the last factor is exemplified in the bottom plot of Figure 2. This shows that tuning BLOCKSIZE by the programmer is a viable approach to

```
int nblks = n/BLOCKSIZE + (n% BLOCKSIZE) ?1 : 0;
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
shared [BLOCKSIZE] double *v = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
double tmp;
int *mythread_indices = (int*)malloc(mythread_nblks * BLOCKSIZE * sizeof(int));
/* let array "mythread_indices" contain random global indices with affinity to "remote threads" */
randomize (mythread_indices, mythread_nblks * BLOCKSIZE);
/* start timing ... */
for (int mb = 0, i = 0; mb < mythread_nblks; mb++)
   for (int k = 0; k < BLOCKSIZE; k++, i++)
      tmp = v[mythread_indices[i]];
/* stop timing ... */
```

Listing 6: A UPC microbenchmark for measuring the latency of individual remote-memory transfers.

Table 2: Time usage (in seconds) of 1000 iterations SpMV for Test problem 1; naive UPC implementation (Listing 2) vs. the first transformed UPC implementation (Listing 3).

	1 thread	2 threads	4 threads	8 threads	16 threads
Naive UPC	895.44	548.57	301.17	173.08	106.10
UPCv1	270.40	159.51	86.37	51.10	28.80

Table 3: Time usage (in seconds) of 1000 iterations SpMV for Test problems 1–3.

	1 node 16 threads	2 nodes 32 threads	4 nodes 64 threads	8 nodes 128 threads	16 nodes 256 threads	32 nodes 512 threads	64 nodes 1024 threads
Test problem 1: 6,810,586 tetrahedrons							
UPCv1	28.80	522.15	443.98	1882.01	551.20	311.54	183.73
UPCv2	39.37	36.70	23.68	18.89	13.61	9.98	9.57
UPCv3	25.01	15.07	8.22	4.65	2.91	2.68	5.56
Test problem 2: 13,009,527 tetrahedrons							
UPCv1	59.14	2525.05	3532.33	3657.95	3078.35	2613.85	1588.67
UPCv2	73.79	69.60	55.33	36.39	24.16	25.06	21.29
UPCv3	46.88	24.97	15.43	10.91	6.25	5.15	7.54
Test problem 3: 25,587,400 tetrahedrons							
UPCv1	115.25	2990.92	1758.94	986.85	1302.52	4653.10	2692.69
UPCv2	154.72	178.14	122.38	81.77	52.99	41.16	44.80
UPCv3	93.30	48.74	26.13	15.37	11.12	7.41	10.16

Table 4: Comparison between actual and predicted time usages (in seconds) of the three transformed UPC implementations for Test problem 1 ($n = 6810586$). The hardware characteristic parameters used for the Abel cluster (16 UPC threads per node) are $W_{\text{thread}}^{\text{private}} = (75/16)$GB/s, $W_{\text{node}}^{\text{remote}} = 6$ GB/s, and $\tau = 3.4\,\mu$s.

THREADS	BLOCKSIZE	$T_{\text{total,actual}}^{\text{UPCv1}}$	$T_{\text{total,predicted}}^{\text{UPCv1}}$	$T_{\text{total,actual}}^{\text{UPCv2}}$	$T_{\text{total,predicted}}^{\text{UPCv2}}$	$T_{\text{total,actual}}^{\text{UPCv3}}$	$T_{\text{total,predicted}}^{\text{UPCv3}}$
16	65536	28.80	26.40	39.37	37.21	25.01	22.95
32	65536	522.15	410.86	36.70	34.30	15.07	14.07
64	65536	443.98	607.08	23.68	20.19	8.22	7.83
128	53200	1882.01	677.99	18.89	12.43	4.65	4.07
256	26600	551.20	679.83	13.61	9.59	2.91	3.06
512	13300	311.54	388.42	9.98	7.83	2.68	2.96
1024	6650	183.73	200.96	9.57	8.15	5.56	3.55

performance optimization. The performance models are essential in this context.

7. Related Work

Many performance studies about UPC programming, e.g., [18–21], selected kernels from the NAS parallel benchmark (NPB) suite [22]. These studies however did not involve irregular fine-grained communication that arises from indirectly indexing the elements of shared arrays. Other published non-NPB benchmarks implemented in UPC, such as reported in [18], had the same limitation. Various UPC implementations of SpMV were studied in [23], but the authors chose to combine a row-wise block

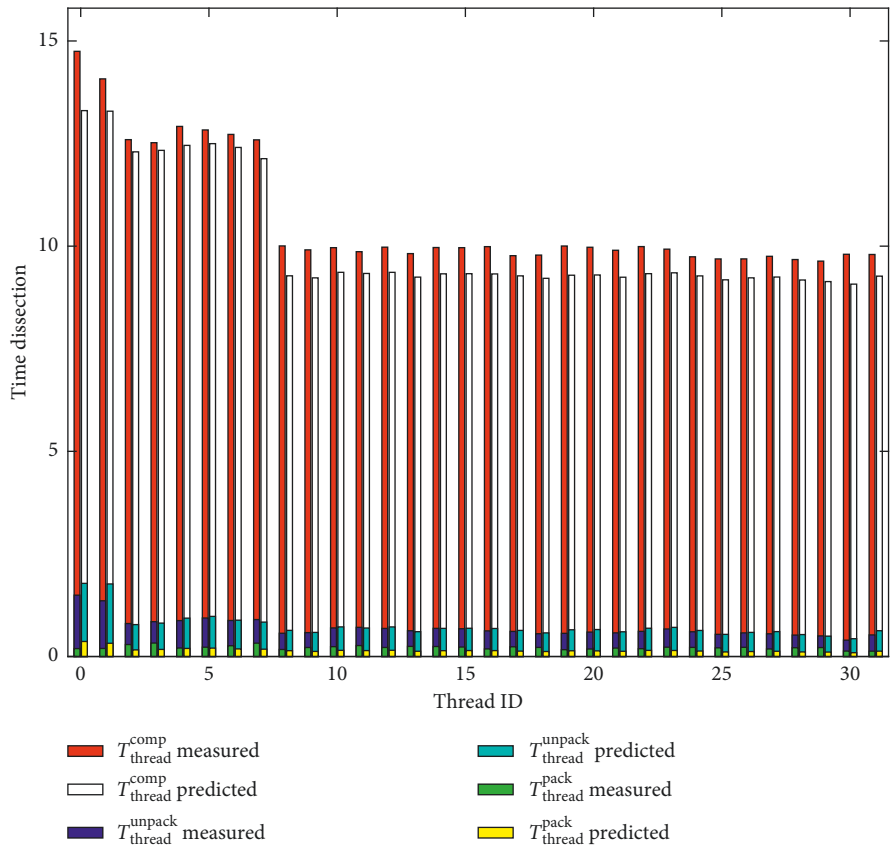
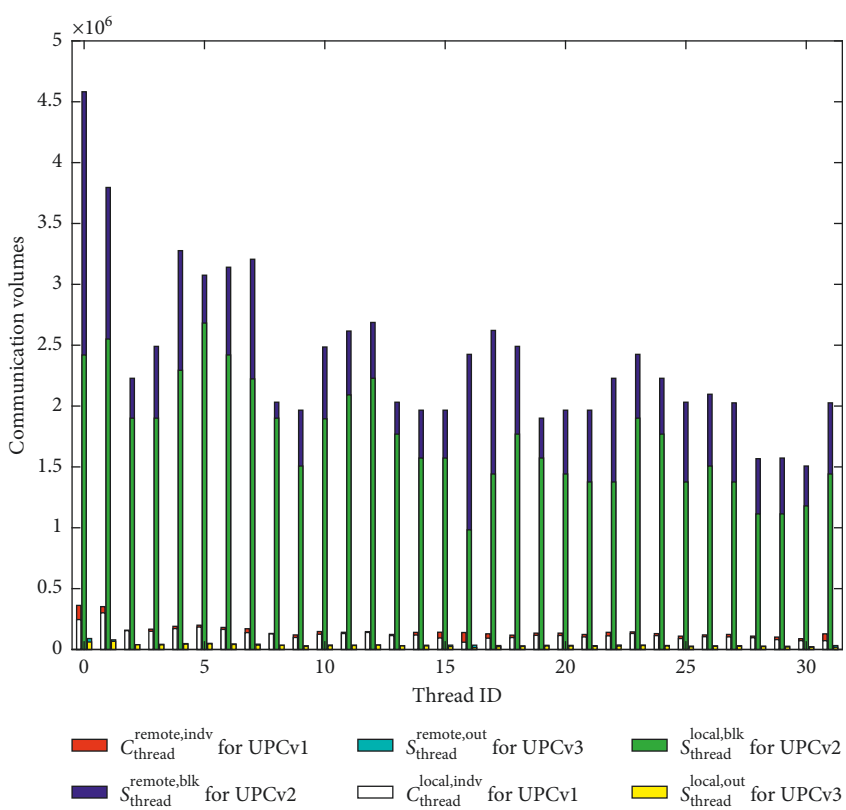


Figure 1: Comparison between per-thread predictions and measurements of T^{comp}_{thread}, T^{unpack}_{thread}, and T^{pack}_{thread} for UPCv3. Test problem 1 (n = 6810586), 32 threads spread over two nodes, with BLOCKSIZE = 65536.



(a)

Figure 2: Continued.

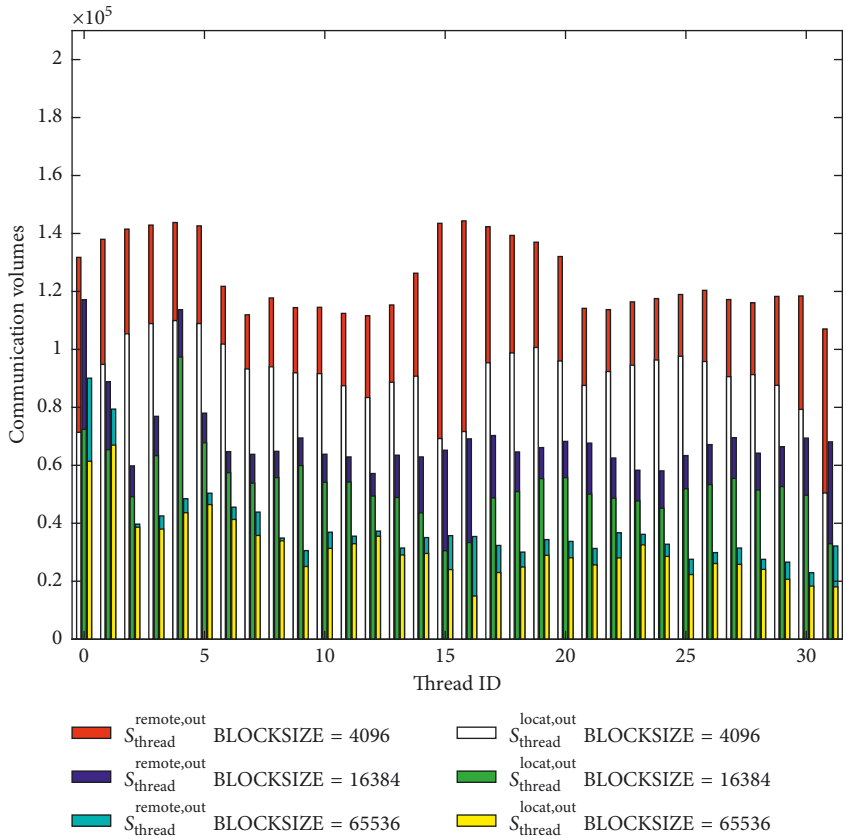


Figure 2: Test problem 1 (n = 6810586), 32 threads spread over two nodes. (a) Per-thread communication volumes required by the three transformed UPC implementations with BLOCKSIZE = 65536. (b) Per-thread communication volumes associated with UPCv3 for different values of BLOCKSIZE.

distribution of the sparse matrix with duplicating the entire source vector x on each thread. Such a UPC implementation of SpMV completely avoided the impact of irregular fine-grained communication, which is the main focus of our present paper. The authors of [24] distributed the source vector x among the UPC threads, but their UPC implementation of SpMV explicitly avoided off-node irregular fine-grained communication, for which the needed values of x were transported in batches from the off-node owner threads using, e.g., upc_memget.

An extended suite of UPC STREAM microbenchmarks, including various scenarios of using pointers-to-shared, was proposed by the authors of [21]. They also reported measurements that clearly reveal the additional overhead due to UPC language features. For our purpose of performance modeling, we only found the so-called "random shared read" microbenchmark defined in [21] to be useful for quantifying τ. This prompted us to write our own microbenchmark (Section 6.2) because we have no access to the source code used for [21].

One approach to alleviating the various types of overhead associated with using pointers-to-shared is by specialized UPC compilers [7, 8, 25, 26]. However, indirectly accessing array elements through pointers-to-shared, which will induce irregular fined-grained communication, cannot be handled by any of the specialized UPC compilers. Manual code transformations are thus necessary, such as those proposed in Sections 4.2 and 4.3 in this paper.

It is quite common that performance studies (such as [18]) and performance models (such as [27]) of UPC programs are built upon "single-value statistics" that is accumulated or averaged over all threads. We see this as an unignorable source of inaccuracy, because considerable variations in the communication volume and pattern may exist between threads, as exemplified by Figures 1 and 2. Ignoring such thread-wise imbalances in communication will lead to inaccurate performance predictions, exactly in the same way as ignoring thread-wise imbalances in computation. At the same time, the performance models proposed in Section 5 of our present paper are kept to a minimalistic style, in that we only rely on four easily benchmarked/obtained hardware characteristic parameters: $W_{\text{thread}}^{\text{private}}$, $W_{\text{node}}^{\text{remote}}$, τ, and the last-level cache line size. This stands as a strong contrast to complicated performance modeling approaches such as in [24].

8. Performance Modeling for a 2D Uniform-Mesh Computation

Our performance modeling strategy in Section 5 was originally derived for the case of fine-grained irregular

communication that is due to indirectly indexed accesses to shared arrays. In this section, we will show that the same methodology, as well as the same hardware characteristic parameters, is applicable to other scenarios. As a concrete example, we will use an *existing* UPC implementation that solves a 2D heat diffusion equation $\partial\phi/\partial t = (\partial^2\phi/\partial x^2) + (\partial^2\phi/\partial y^2)$ on a uniform mesh. The UPC code was kindly provided by Dr. Rolf Rabenseifner at HLRS, in connection with a short course on PGAS programming [28].

8.1. Brief Description of the Code

8.1.1. Data Structure. The global 2D solution domain is rectangular, so the UPC threads are arranged as a 2D processing grid, with mprocs rows and nprocs columns (note THREADS equals mprocs ∗ nprocs). Each thread is thus identified by an index pair (iproc,kproc), where iproc = MYTHREAD/nprocs and kproc = MYTHREAD% nprocs. The global 2D domain, of dimension $M \times N$, is evenly divided among the threads. Each thread is responsible for a 2D subdomain of dimension $m \times n$, which includes a surrounding halo layer needed for communication with the neighboring threads. The main data structure consists of the following components:

```
shared[] double * shared xphi[THREADS];
double *phin;
xphi[MYTHREAD] = (shared[]double*)
upc_alloc(m * n * sizeof(double));
phin = (double*) malloc(m * n * sizeof(double));
```

Note that the values in each shared array xphi [MYTHREAD] have affinity to the allocating thread but are accessible by all the other threads. Each thread also allocates a private array named phin to store its portion of the numerical solution for a new time level, to be computed based on the numerical solution for the previous time level, which is assumed to reside in xphi [MYTHREAD].

8.1.2. Halo Data Exchange. The halo data exchange, which is needed between the neighboring threads, is realized by that every thread calls upc_memget on each of the four sides (if a neighboring thread exists). In the vertical direction, the values to be transfered from the upper and lower neighbors already lie contiguously in the memory of the owner threads. There is thus no need to explicitly pack the messages. In the horizontal direction, however, message packing is needed before upc_memget can be invoked towards the left and right neighbors. The following additional data structure is needed with packing and unpacking the horizontal messages:

```
/* scratch arrays for UPC halo exchange of non-
contiguous data */
shared[] double * shared xphivec_coord1first[THREADS];
shared[] double * shared xphivec_coord1last [THREADS];
double *halovec_coord1first, *halovec_coord1last;
xphivec_coord1first[MYTHREAD] = (shared[] double*)
upc_alloc((m - 2) * sizeof(double));
xphivec_coord1last[MYTHREAD] = (shared[] double*)
upc_alloc((m - 2) * sizeof(double));
halovec_coord1first = (double*) malloc((m - 2) * sizeof
(double));
halovec_coord1last = (double*) malloc((m - 2) * sizeof
(double));
```

Consequently, the function for executing the halo data exchange is implemented as follows:

8.1.3. Computation. The computation that is carried out at each time level can be seen in the following time loop:

8.2. Performance Modeling. By slightly changing the formulas in Section 5.2.5, we can model the cost of the different parts involved in the function halo_exchange_intrinsic (Listing 7) as follows:

$$T_{\text{thread}}^{\text{halo,pack}} = T_{\text{thread}}^{\text{halo,unpack}} = \frac{\left(S_{\text{thread}}^{\text{local,horiz}} + S_{\text{thread}}^{\text{remote,horiz}}\right)(\text{sizeof}(\text{double}) + \text{sizeof}(\text{cache line}))}{W_{\text{thread}}^{\text{private}}}, \tag{20}$$

$$T_{\text{node}}^{\text{halo,memget}} = \max_{\forall\text{threads in node}} \frac{2 \cdot S_{\text{thread}}^{\text{local}} \cdot \text{sizeof}(\text{double})}{W_{\text{thread}}^{\text{private}}} + \sum_{\forall\text{threads in node}} \left(C_{\text{thread}}^{\text{remote}} \cdot \tau + \frac{S_{\text{thread}}^{\text{remote}} \cdot \text{sizeof}(\text{double})}{W_{\text{node}}^{\text{remote}}}\right). \tag{21}$$

Comparing (20) with (12) from Section 5.2.5, we can see that the cost due to indirect array indexing (sizeof(int)) is no longer applicable. We have also assumed in (20) that reading/writing from/to noncontiguous locations in the array phi each costs a cache line. The value of $S_{\text{thread}}^{\text{local,horiz}}$ in (20) denotes the total volume of local and remote messages, per thread, to be transfered in the horizontal direction. This can be precisely calculated when the values of m and n, as well as the thread grid layout, are known. Formula (21) is essentially the same as (13) from Section 5.2.5. It is commented that $S_{\text{thread}}^{\text{local}}$ in (21) denotes the total volume of all local messages (in both horizontal and vertical directions) per thread, likewise denotes $S_{\text{thread}}^{\text{remote}}$ the total volume of all remote messages, with $C_{\text{thread}}^{\text{remote}}$ denoting the number of remote messages per thread. Putting them together, the total time spent on halo_exchange_intrinsic is modeled as

```
#define idx(i, k) ((i) * n + (k))
#define rank(ip, kp) ((ip) * nprocs + (kp))
void halo_exchange_intrinsic()
{
   double* phi = (double*) xphi[MYTHREAD];
   int i, k;
   /* packing messages for the horizontal direction */
   if (kproc > 0) {
      double *phivec_coord1first = (double*) xphivec_coord1first[MYTHREAD];
      for (i = 0; i < m − 2; i++)
         phivec_coord1first[i] = phi[idx(i + 1, 1)];
   }
   if (kproc < nprocs − 1) {
      double *phivec_coord1last = (double*) xphivec_coord1last[MYTHREAD];
      for (i = 0; i < m − 2; i++)
         phivec_coord1last[i] = phi[idx(i + 1, n − 2)];
   }
   upc_barrier;
   /* message transfer and unpacking (needed for the horizontal direction) */
   if (kproc > 0) {
      upc_memget(halovec_coord1first, xphivec_coord1last[rank(iproc, kproc − 1)], (m − 2) * sizeof(double));
      for (i = 1; i < m − 1; i++)
         phi[idx(i, 0)] = halovec_coord1first[i − 1];
   if (kproc > nprocs − 1) {
      upc_memget(halovec_coord1last, xphivec_coord1first[rank(iproc,kproc + 1)], (m − 2) * sizeof(double));
      for (i = 1; i < m − 1; i++)
         phi[idx(i, n − 1)] = halovec_coord1last[i − 1];
   }
   if (iproc > 0)
      upc_memget(&phi[idx(0, 1)], &(xphi[rank(iproc − 1, kproc)][idx(m − 2, 1)]), (n − 2) * sizeof(double));
   if (iproc < mprocs − 1)
      upc_memget(&phi[idx(m − 1, 1)], &(xphi[rank(iproc + 1, kproc)][idx(1, 1)]), (n − 2) * sizeof(double));
}
```

Listing 7: The halo data exchange function of an existing 2D heat equation solver.

```
#define idx(i, k) ((i) * n + (k))
for (it = 1; it ≤ itmax; it++) {
   double *phi = (double*) xphi[MYTHREAD];
   int i, k;
   /* communication per time step */
   halo_exchange_intrinsic();
   /* computation per time step */
   for (i = 1; i < m − 1; i++)
      for (k = 1; k < n − 1; k++)
         phin[idx(i, k)] = ((phi[idx(i + 1, k)] + phi[idx(i − 1, k)] − 2 * phi[idx(i, k)]) * dy2i + (phi[idx(i, k + 1)] + phi[idx(i, k − 1)] − 2 * phi
[idx(i, k)]) * dx2i) * dt;
   /* copying the content of phin to phi, checking convergence etc. */
   // . . .
}
```

Listing 8: The computational kernel of an existing 2D heat equation solver.

$$T_{2D}^{halo} = \max_{\forall nodes}\left(\left(\max_{\forall threads\ in\ node} T_{thread}^{halo,pack}\right) + T_{node}^{halo,memget} + \left(\max_{\forall threads\ in\ node} T_{thread}^{halo,unpack}\right)\right). \quad (22)$$

The time spent on computation during each time step (Listing 8) is modeled as

$$T_{2D}^{comp} = \frac{3(m-2)(n-2)\cdot \text{sizeof}(\text{double})}{W_{thread}^{private}}. \quad (23)$$

Table 5: Comparison of actual and predicted time usages when running 1000 time steps of the 2D heat equation solver. The same hardware characteristic parameters as in Table 4 are used.

THREADS	Partitioning	$T^{\text{halo}}_{\text{2D,actual}}$	$T^{\text{halo}}_{\text{2D,predicted}}$	$T^{\text{comp}}_{\text{2D,actual}}$	$T^{\text{comp}}_{\text{2D,predicted}}$
Mesh size 20000×20000					
16	4×4	0.52	0.33	122.53	122.07
32	4×8	0.44	0.37	61.55	61.04
64	8×8	0.27	0.21	30.78	30.52
128	8×16	0.29	0.21	15.31	15.26
256	16×16	0.18	0.13	7.70	7.63
512	16×32	0.14	0.14	3.85	3.81
Mesh size 40000×40000					
16	4×4	1.55	0.65	489.96	488.28
32	4×8	1.08	0.73	246.25	244.14
64	8×8	0.64	0.42	122.82	122.07
128	8×16	0.64	0.42	61.85	61.04
256	16×16	0.42	0.26	31.01	30.52
512	16×32	0.29	0.26	15.47	15.26

Here we have assumed that every thread has exactly the same amount of computational work. The value of $T^{\text{comp}}_{\text{2D}}$ is estimated on the basis of the minimum amount of traffic between the memory and the last-level cache [29].

Table 5 shows a comparison between the actual time usages of the 2D UPC solver with the predicted values of $T^{\text{halo}}_{\text{2D}}$ and $T^{\text{comp}}_{\text{2D}}$. We have used the same hardware characteristic parameters as in Table 4. It can be seen that the predictions of $T^{\text{comp}}_{\text{2D}}$ agree excellently with the actual measurements, while the prediction accuracy of $T^{\text{halo}}_{\text{2D}}$ is on average 72%.

9. Conclusion

Our starting point is a naive UPC implementation of the SpMV $y = Mx$ in Section 3.2. This naive implementation excessively uses shared arrays, pointers-to-shared, and upc_forall. We have developed three increasingly aggressive code transformations in Section 4 aiming at performance enhancement. The transformations include explicit thread privatization that avoids upc_forall and casts pointers-to-shared to pointers-to-local whenever possible, as well as removing fine-grained irregular communications that are implicitly caused by indirectly indexed accesses to the shared array x. The latter transformation is realized by letting each thread adopt a private mythread_x_copy array that is prepared with explicit one-sided communications (using two different strategies) prior to the SpMV computation. Numerical experiments of a realistic application of SpMV and the associated time measurements reported in Section 6 have demonstrated the performance benefits due to the code transformations. The performance benefits are also justified and quantified by the three performance models proposed in Section 5.

While the code transformations lead to improved performance, the complexity of UPC programming is increased at the same time (trading programmability for performance is by no means specific for our special SpMV example; the textbook of UPC [5] has ample examples). The naive UPC implementation in Section 3.2 shows the easy programmability of UPC that is fully comparable with OpenMP, as discussed in [30]. The first code transformation, in form of explicit thread privatization shown in Section 4.1, may be done by automated code translation. The second and third code transformations, see Sections 4.2 and 4.3, are however more involved. The adoption of one mythread_x_copy array per thread also increases the memory footprint. Despite reduced programmability, all the UPC implementations maintain some advantages over OpenMP in targeting distributed-shared memory systems and promoting data locality. The third code transformation, UPCv3, results in a programming style quite similar to that of MPI. Nevertheless, UPCv3 is easier to code than MPI, because global indices are retained for accessing the entries of array mythread_x_copy. An MPI counterpart, where all arrays are explicitly partitioned among processes, will have to map the global indices to local indices. Moreover, one-sided explicit communications via UPC upc_memget and upc_memput functions are easier to use. Performance advantage of UPC's one-sided communication over the MPI counterpart has also been reported in [31]. On the other hand, persistent advantages of MPI over UPC include better data locality and more flexible data partitionings. A comparison of performance and programmability between UPC and MPI is given in [32] for a realistic fluid dynamic implementation. For a general comparison between OpenMP, UPC, and MPI programming, we refer to [20].

It should be stressed that the SpMV computation is chosen for this paper as an illustrating example of fine-grained irregular communication that may arise in connection with naive UPC programming. The focus is not on the SpMV itself, but on the code transformations and the performance models in general. Moreover, our performance models are to a great extent independent of the UPC programming details, but rather focusing on the incurred communication style, volume, and frequency. The hardware characteristic parameters $W^{\text{private}}_{\text{thread}}$, $W^{\text{remote}}_{\text{node}}$, and the cache line size are equally applicable to similar communications and memory-bandwidth bound computations implemented by other programming models rather than UPC. Even the latency of individual remote-memory accesses, τ, can alternatively be measured by a standard MPI ping-pong

benchmark. Our philosophy is to represent a target hardware system by only four characteristic parameters, whereas the accuracy of the performance prediction relies on an accurate counting of the incurred communication volumes and frequencies. Accurate counting is essential and thus cannot be generalized, because different combinations of the problem size, number of threads, and block size will almost certainly lead to different levels of performance for the same parallel implementation.

Data Availability

The data used to support the findings of this study are included in the tables and figures within the article.

Disclosure

The current affiliation of Martina Prugger is Vanderbilt University, 2201 West End Ave, Nashville, TN 37235, USA.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was performed on hardware resources provided by UNINETT Sigma2—the National Infrastructure for High Performance Computing and Data Storage in Norway, via Project NN2849K. The work was supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 671657, the European Union Seventh Framework Programme (grant no. 611183), and the Research Council of Norway (grant nos. 231746/F20, 214113/F20, and 251186/F20). Martina Prugger was supported by the VSC ResearchCenter funded by the Austrian Federal Ministry of Science, Research and Economy (bmwfw) and by the Doctoral Programme Computational Interdisciplinary Modelling (DK CIM) at the University of Innsbruck.

References

[1] G. Almasi, "PGAS (partitioned global address space) languages," in *Encyclopedia of Parallel Computing*, D. Padua, Ed., pp. 1539–1545, Springer, Berlin, Germany, 2011.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein et al., "Parallel programming in split-C," in *Proceedings of 1993 ACM/IEEE Conference on Supercomputing (Supercomputing'93)*, pp. 262–273, Portland, OR, USA, November 1993.

[3] M. de Wael, S. Marr, B. de Fraine, T. van Cutsem, and W. de Meuter, "Partitioned global address space languages," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–27, 2015.

[4] PGAS—Partitioned Global Address Space, 2016, http://www.pgas.org.

[5] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, John Wiley & Sons, Hoboken, NJ, USA, 2005.

[6] UPC Consortium, "UPC language specifications version 1.3," 2013, http://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf.

[7] W.-Y. Chen, *Optimizing partitioned global address space programs for cluster Architectures*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, USA, 2007.

[8] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained UPC applications," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, St. Louis, MO, USA, September 2005.

[9] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 User's Guide," Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Austin, TX, USA, 1979.

[10] Y. Zheng, "Optimizing UPC programs for multi-core systems," *Scientific Programming*, vol. 18, no. 3-4, pp. 183–191, 2010.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[12] J. Langguth, N. Wu, J. Chai, and X. Cai, "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 120–131, 2015.

[13] J. D. McCalpin, "STREAM: sustainable memory bandwidth in high performance computers," Technical Report, University of Virginia, Charlottesville, VA, USA, 2007.

[14] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous CPU-GPU computations for unstructured tetrahedra l meshes," *IEEE Micro*, vol. 35, no. 4, pp. 6–15, 2015.

[15] H. Si, "TetGen, a delaunay-based quality tetrahedral mesh generator," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, pp. 1–36, 2015.

[16] The Abel computer cluster, 2018, https://www.uio.no/english/services/it/research/hpc/abel/.

[17] Berkeley UPC—Unified Parallel C, 2018, http://upc.lbl.gov.

[18] C. Barton, C. Caşcaval, and J. N. Amaral, "A characterization of shared data access patterns in UPC programs," in *Proceedings of 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, vol. 4382, pp. 111–125, Springer, New Orleans, LA, USA, November 2006.

[19] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: a NPB experimental study," in *Proceedings of ACM/IEEE SC 2002 Conference (SC'02)*, Baltimore, MD, USA, November 2002.

[20] H. Shan, F. Blagojević, S.-J. Min et al., "A programming model performance study using the NAS parallel benchmarks," *Scientific Programming*, vol. 18, no. 3-4, pp. 153–167, 2010.

[21] Z. Zhang and S. R. Seidel, "Benchmark measurements of current UPC platforms," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, April 2005.

[22] D. Bailey, E. Barszcz, J. Barton et al., "The NAS parallel benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, Mountain View, CA, USA, 1994.

[23] J. González-Domínguez, Ó. García-López, G. L. Taboada, M. J. Martín, and J. Touriño, "Performance evaluation of sparse matrix products in UPC," *Journal of Supercomputing*, vol. 64, no. 1, pp. 100–109, 2013.

[24] S. Li, C. Hu, J. Zhang, and Y. Zhang, "Automatic tuning of sparse matrix-vector multiplication on multicore clusters," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–14, 2015.

[25] M. Alvanos, *Optimization techniques for fine-grained communication in PGAS environments*, Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2013.

[26] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast address translation techniques for distributed shared memory compilers," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, April 2005.

[27] Z. Zhang and S. R. Seidel, "A performance model for fine-grain accesses in UPC," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodes Island, Greece, April 2006.

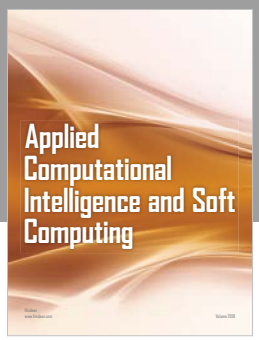
[28] R. Rabenseifner, *Short Course: Introduction to Unified Parallel C (UPC) and Co-array Fortran (CAF)*, HLRS, University of Stuttgart, Stuttgart, Germany, 2015.

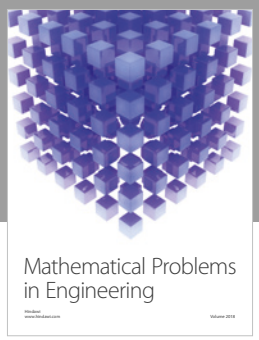
[29] H. Stengel, T. Jan, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 207–216, ACM, Irvine, CA, USA, June 2015.

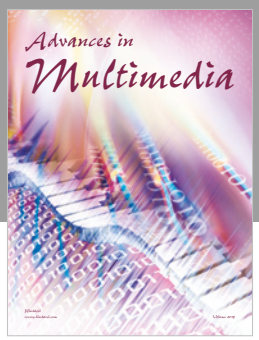
[30] A. Marowka, "Execution model of three parallel languages: OpenMP, UPC and CAF," *Scientific Programming*, vol. 13, no. 2, pp. 127–135, 2005.

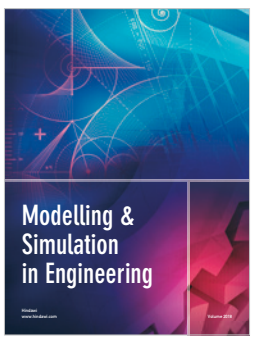
[31] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick, "An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*, pp. 1115–1125, IEEE, Phoenix, AZ, USA, May 2014.

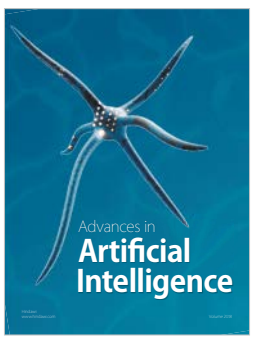
[32] M. Prugger, L. Einkemmer, and A. Ostermann, "Evaluation of the partitioned global address space (PGAS) model for an inviscid Euler solver," *Parallel Computing*, vol. 60, pp. 22–40, 2016.



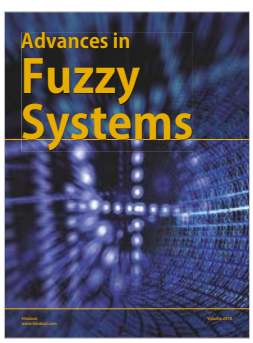
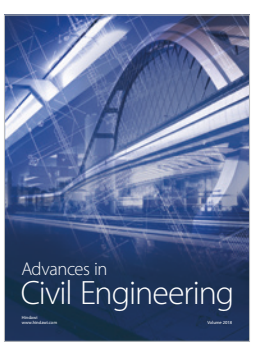
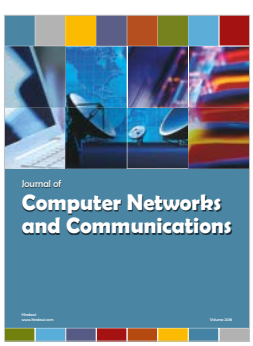
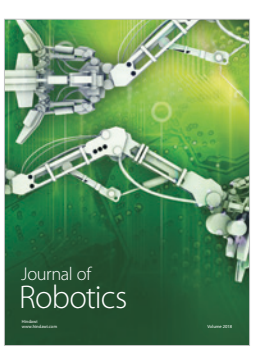
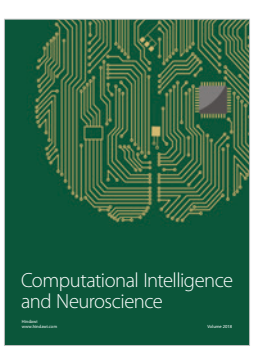








Hindawi

Submit your manuscripts at

www.hindawi.com