Paper accepted at IPDPS'05 (Main Conference)

FAST ADDRESS TRANSLATION TECHNIQUES FOR DISTRIBUTED SHARED MEMORY COMPILERS

François Cantonnet [1], Tarek A. El-Ghazawi [1], Pascal Lorenz [2], Jaafer Gaber [3]
{ fcantonn, tarek }@gwu.edu, lorenz@ieee.org, gaber@utbm.fr
[1]: Department of Electrical and Computer Engineering, The George Washington University, USA
[2]: Université de Haute Alsace, FRANCE, [3]: Université de Technologie de Belfort-Montbéliard, FRANCE

Abstract

The Distributed Shared Memory (DSM) model is designed to leverage the ease of programming of the shared memory paradigm, while enabling the high-performance by expressing locality as in the message-passing model. Experience, however, has shown that DSM programming languages, such as UPC, may be unable to deliver the expected high level of performance. Initial investigations have shown that among the major reasons is the overhead of translating from the UPC memory model to the target architecture virtual addresses space, which can be very costly. Experimental measurements have shown this overhead increasing execution time by up to three orders of magnitude. Previous work has also shown that some of this overhead can be avoided by hand-tuning, which on the other hand can significantly decrease the UPC ease of use. In addition, such tuning can only improve the performance of local shared accesses but not remote shared accesses. Therefore, a new technique that resembles the Translation Look Aside Buffers (TLBs) is proposed here. This technique, which is called the Memory Model Translation Buffer (MMTB) has been implemented in the GCC-UPC compiler using two alternative strategies, full-table (FT) and reduced-table (RT). It will be shown that the MMTB strategies can lead to a performance boost of up to 700%, enabling ease-of-programming while performing at a similar performance to hand-tuned UPC and MPI codes.

1. Introduction

UPC, also known as Unified Parallel C, is an explicit parallel programming language based on the ISO C standard [ISO99] and the distributed shared memory programming model. Alternatively, the distributed shared memory programming model is also known as the Partitioned Global Address Space (PGAS) model. UPC capitalizes on the experience gained from its predecessor distributed shared memory C compilers such as Split-C[CUL93], AC[CAR99] and PCP[BRO95]. UPC maintains the C philosophy of making programs concise while keeping the programmer closer to the hardware to gain more performance. UPC is becoming widely accepted in the high-performance computing community. The UPC Specifications v1.0 document was produced in February 2001, and a new version v1.1.1, was released in October 2003 [ELG03] by the UPC consortium. UPC is developed based on the lessons learned from message passing and the pure shared memory programming paradigm. The shared memory model brings ease of use, as in this style of programming remote memory accesses need not to be expressed explicitly. However, loss of performance can arise from remote accesses. Message passing offers explicit distribution of data and work such that unnecessary messages can be avoided. On the other hand, message passing is characterized by its significant overhead for small messages and it is largely hard to use. The distributed shared memory programming paradigm, as in UPC, brings the best of these two worlds. It allows the exploitation of data locality by distributing data and processing under the control of the programmer, while maintaining ease of use.

The Non-Uniform Memory Access Architecture (NUMA) offers global address space, distributed among the nodes. This architecture can therefore provide a way to reduce overhead for implementing distributed shared memory programming paradigms, such as UPC. In this study, it will be shown that there is still significant room for the current UPC compiler implementations to take more advantage of the capabilities offered by this architecture. In general, it will be shown that in such architectures, the UPC pointer addresses can be resolved easier by caching the shared pointer values along with the corresponding virtual addresses in the global shared space, using a form of translation look-aside buffers, referred to here as the memory model translation buffers (MMTB).

This paper is organized as follows. Section 2 gives a brief description of the UPC language, while section 3 introduces the experimental testbed and workloads used. Section 4 presents the Memory Model Translation Buffers (MMTB) mechanism. Section 5 gives performance measurements, followed by conclusions in Section 6.

2. Unified Parallel C (UPC)

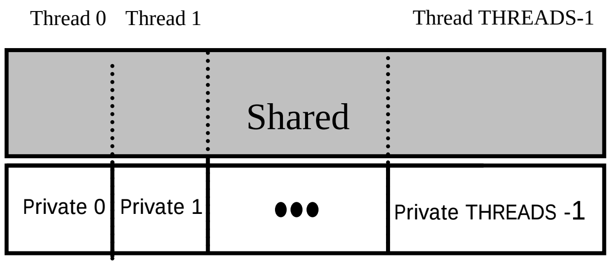


Figure 1 The UPC Memory and Execution Model

Figure 1 illustrates the memory and execution model as viewed by UPC codes and programmers. Under UPC, memory is composed of a shared memory space and a private memory space. A number of threads work independently and each of them can reference any address in the shared space, but only its own private space. The total number of threads is **`THREADS`** and each thread can identify itself using **`MYTHREAD`**, where **`THREADS`** and **`MYTHREAD`** can be seen as special constants. The shared space, however, is logically divided into partitions each with a special association (affinity) to a given thread. The idea is to make UPC enable the programmers, with proper declarations, to keep the shared data that will be dominantly processed by a given thread associated with that thread. Thus, a thread and the data that has affinity to it can likely be mapped by the system into the same physical node.

Since UPC is an explicit parallel extension of ISO C, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control of the distribution of data across the threads. Among the interesting and complementary UPC features is a work-sharing iteration statement, known as `upc_forall`. This statement helps to distribute independent loop iterations across the threads, such that iterations and data that are processed by them are assigned to the same thread. UPC also defines a number of rich private and shared pointer concepts. In addition, UPC supports dynamic shared memory allocations. There is generally no implicit synchronization in UPC. Therefore, the language offers a rich range of synchronization and memory consistency control constructs. Among the most interesting synchronization concepts is the non-blocking barrier, which allows overlapping local computations with thread synchronization.

3. Testbed and Workloads

3.1 TESTBED

As an experimental testbed, the SGI Origin 2000 was used. This machine is based on the NUMA (Non Uniform Memory Access) architecture model, with a global address space single system image (SSI). The Origin 2000 has 32 MIPS R10000 processors, each running at 195MHz, with 2 processors per node. The total memory available is 8GB with a bandwidth of 780MB/sec. The Origin system is running the Irix 6.5 Operating System.

The SGI Origin has the GCC-UPC v3.2.3.5 compiler [INT04] installed, based on the GCC compiler, v3.2, but with added front-end module and runtime environment for UPC. The low-level monitoring is done using the SGI Speedshop software suite.

3.2 WORKLOADS

The results with and without the proposed improvements exposed in this paper were examined considering common benchmarking workloads, like STREAM benchmark and the GUPS benchmark. The UPC STREAM benchmark [MCC95][ELG01b] is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. Giga-Updates Per Second (GUPS), unlike the regular accesses of Stream, is a simple program performing updates to random locations in a large shared array [GAE02][CAN03]. Microkernels [ELG01a] such as the Sobel Edge Detector, Matrix Multiplication problem as well as workloads from the NAS Parallel Benchmark (NPB) Suite [NPB02] are considered. In addition, measurements for a MPI implementation, as well as hand-tuned UPC implementation, are given for many of the workloads as performance references. These workloads can be downloaded from the UPC website (http://upc.gwu.edu), except for the NPB MPI suite, which is directly taken from the NAS website. In this paper, the focus will be on the five NPB kernels, with a considerable problem size, as defined in class C.

4. Enhancing the Compiler with the Memory Model Translation Buffer Mechanism

One rich feature of UPC is that shared arrays can be distributed among the threads in blocks of arbitrary (user defined) sizes. This requires that compilers provide the translation between the UPC memory model and the real layout in the virtual address space. It is shown in this work that mostly all current UPC compilers are poorly implemented as far as the memory model address translation is concerned. Further, optimization techniques for compilers are defined and demonstrated to do such translation efficiently.

4.1 STREAM RESULTS

	MB/Sec	**Bulk Operations**			**Element-by-Element Operations**			
		Memory copy	**Block Get**	**Block Put**	**Array Set**	**Array Copy**	**Sum**	**Scale**
C	*GCC*	127	N/A	N/A	175	106	223	108
	UPC Private	127	N/A	N/A	173	106	215	107
UPC	*UPC Local Shared*	139	140	136	26	14	31	13
	UPC Shared (SMP)	130	129	136	26	13	30	13
	UPC Shared (Remote)	112	117	136	24	12	28	12

Table 1 STREAM Benchmark on the SGI Origin 2000 (Bandwidths in MB/sec)

Table 1 presents the results of the STREAM micro-benchmark. The GCC sequential behavior is used as a baseline of the best performance available on this architecture. The UPC private indicates local-shared accesses that are done through a regular private C pointer. These are showing the same performance as GCC. These measurements, collected on the Origin 2000, show that UPC bulk remote shared accesses can be slower than UPC local shared accesses, which is expected. However, the slight difference between the two indicates that there could be some common overhead. On the other hand, UPC single (element-by-element) local shared accesses can be up to an order of magnitude slower than private accesses. This is however inconsistent with intuition as both local-shared and private spaces are likely co-located in the same physical node and should exhibit the same bandwidth. It is concluded, however, that this inconsistency is mainly due to the fact that UPC compilers are not efficiently translating from the UPC memory model space to the virtual space addresses. Meanwhile, bulk operations (memory copy, block get and block put) are taking advantage of block transfers, as they can amortize overhead better than small accesses.

4.2 OVERHEAD IN SHARED MEMORY ACCESS

In order to confirm the belief that the memory model translation to virtual space is costly, an investigation was conducted to quantify this overhead experimentally. Figure 2 illustrates the different overhead sources for any shared access, namely a function call to the get or put, that reside in the UPC Runtime system and then the translation from the UPC memory model to the shared virtual space. Studies have shown that a single local shared read causes about 20 memory reads and 6 memory writes [CAN03], as detected by a low-level profiling facility in the SGI architecture.

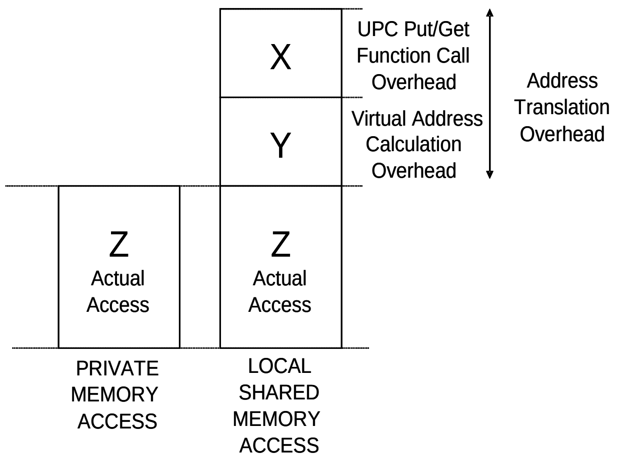

X
UPC Put/Get Function Call Overhead
Y
Virtual Address Calculation Overhead
Address Translation Overhead
Z
Actual Access
Z
Actual Access
PRIVATE MEMORY ACCESS
LOCAL SHARED MEMORY ACCESS

Figure 2 Overheads Present in Local-Shared Memory Accesses

To quantify the amount of time spent in each step X, Y and Z as per Figure 2, a small program which performs an array set, into either a shared or private array on element-by-element basis is used. Using this program, the time to set a complete private array was measured and the average memory access time, Z, was computed. The time X was estimated by introducing a dummy function in the UPC runtime system and calling it. The address calculation overhead Y was deduced from the overall execution time X+Y+Z while accessing a local shared array. The corresponding measurements observed are given in Figure 3.

From this figure, it is clear that the address translation overhead is quite significant as it added more than 70% of overhead work in this case. This also demonstrates the real need for optimization.

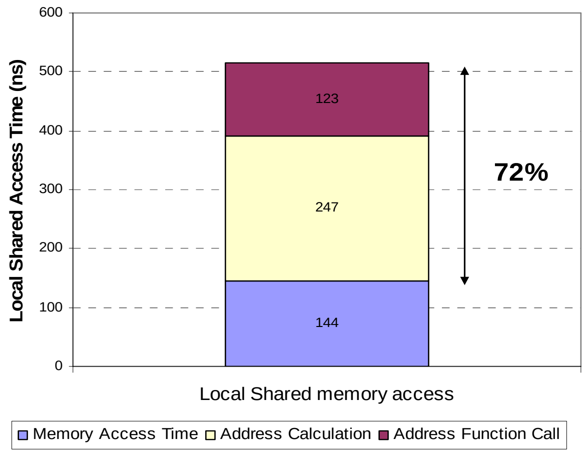


Figure 3 Quantification of the Address Translation Overheads

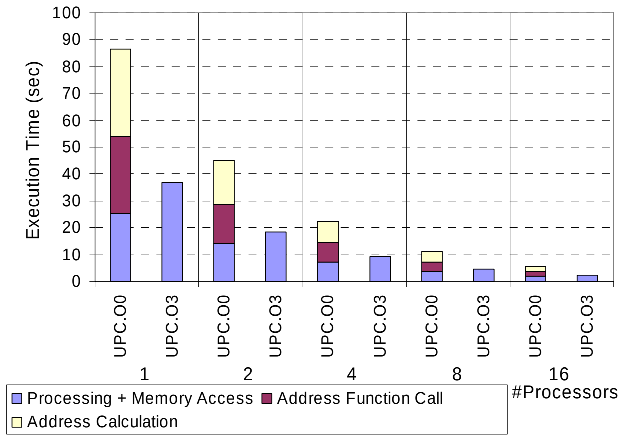


Figure 4 Performance of Regular UPC and Hand-tuned UPC for Sobel Edge Detection

Figure 4 illustrates the amount of time spent on computation as well as in the address translation for the Sobel Edge Detection kernel. Two UPC codes are compared, the non-optimized UPC code, referred as UPC.O0, and a hand-optimized version of the same code, called UPC.O3. The UPC Ox notations indicates the amount of manual optimizations done on the code to emulate the compiler automatic optimizations [ELG01b]. O0 denotes a 'plain' UPC code, in which no hand-optimization have been done, whereas O3 denotes a hand-optimized UPC code, in which privatization, aggregation of remote accesses as well as prefetching have been manually implemented. Space-privatization refers to accessing local shared data with private pointers, to avoid the address translation overhead. This lowered significantly the execution time of Sobel by roughly a factor of 3, comparing the execution time of UPC.O0 and UPC.O3. One important remark is that in Sobel Edge, the address translation overhead is taking 2/3 of the total execution time. Other applications have shown even larger overhead.

4.3 REDUCING THE MEMORY MODEL ADDRESS TRANSLATION COST WITH LOOK-UP BUFFERS

In this section, a mechanism to considerably decrease the memory model translation overhead is introduced. This optimization can be integrated into the UPC compiler to increase the performance of pure UPC without the need for hand optimizations. As a proof of concept, the technique was introduced into the GCC-UPC compiler from Intrepid. This technique uses look-up tables that are called here the memory model translation buffers (MMTB). The main idea is to, first avoid doing any function calls to the UPC Runtime system and secondly, reduce the number of computations needed for address calculation. Thus, instead of computing the virtual address for each shared memory reference at each access, a look-up table is used for caching the translation to the virtual address in a manner similar to the translation look-aside buffer in virtual memory management. Thus, the user can get better performance, due to the significantly reduced overhead without sacrificing the ease of programming of UPC.

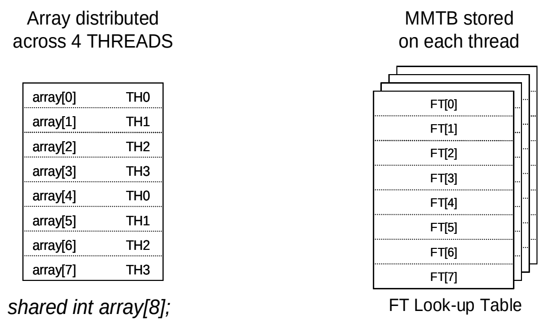


Figure 5 FT MMTB look-up with Full Tables

Consider shared [B] int array[32];

<u>To Initialize FT:</u>
  $\forall i \in [0,31]$,
  FT[i] = _get_vaddr(&array[i]);

<u>To Access array[ ]:</u>
  $\forall i \in [0,31]$,
  array[i] = _get_value_at(FT[i]);

Figure 6 Example Initializing and Using the MMTB FT Mechanism

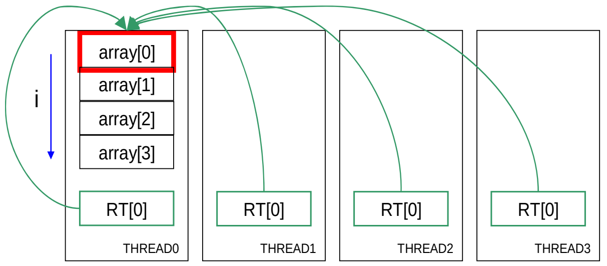
In this work, two alternative methods to implement this MMTB concept were developed. These two alternative methods are referred to in this paper as the Full Table (FT) and the Reduced Table (RT) methods. While these methods increase the memory requirements of a given application, it will be shown that the performance becomes close to that of the hand-tuned UPC code.

The first strategy, FT, is straight-forward. A private table of pointers, of the same size as a given data array is created in the private space of each thread and is filled with all the virtual addresses of the shared array in the local system, Figure 5. Virtual addresses of a shared object reference are obtained through a low-level function which is machine dependant. Thus, each thread has an identical copy of the whole address translation table, in its private space. This table is used to access the data of the array, providing a faster access since there is no required address calculation. In addition, for local-shared memory references, privatization is no longer needed since the overhead is now reduced. Figure 5 shows an example of how to initialize and use the MMTB FT method. Two low-level functions, `_get_vaddr()` and `_get_value_at()` for getting the virtual address of a shared element and to dereference a virtual address respectively have been implemented in the UPC Runtime system. The strength of this method is its simplicity and its direct mapping between the UPC memory model and the virtual space. However, it requires significant memory space. It also requires significant memory accesses as it generates one private memory read per shared memory access, which can lead to a competition over the machine resources with the UPC program. Note that even this is still a substantial improvement over the current implementation.

The second mechanism RT for the memory model translation buffer is an enhancement of FT, which tends to reduce the amount of memory required for the MMTB table. RT takes advantage of the fact that the elements of each block in a shared array is required by UPC to be contiguous in the virtual space. In addition, all blocks of a shared array that have affinity to a given thread are also required to be contiguous. Thus for each shared array, a private table is created but with a factor of "blocksize" less than the FT scheme, since this table holds only the virtual address of each block starting address. This RT method provides a tradeoff between extra address calculations and the amount of memory space required.

For further improvement of the RT method, three cases are distinguished, based on the blocksize of the shared data array:

- Infinite blocksize (expressed as [ ]), as illustrated Figure 7
- Default blocksize of one element per thread (expressed as [1]), Figure 8
- Arbitrary blocksize, which consists of all other cases, Figure 9



Only first element of the array needs to be saved, since all array data are contiguous

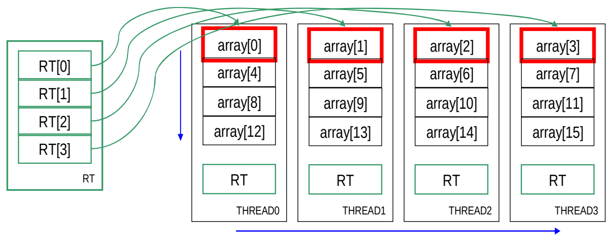
Figure 7a MMTB RT Mechanism with Infinite Blocked Shared Arrays

```
Consider shared [ ] int array[4];

To Initialize RT:
  RT[0]= _get_vaddr(array[0]);

To Access array[ ]:
  ∀i ∈ [0,3],
  array[i] = _get_value_at(RT[0] + i);
```

Figure 7b Example Initializing and Using the MMTB RT Mechanism with Infinite Blocked Shared Arrays

Figure 7 Example of MMTB with RT, Using Infinite Blocked Shared Array and 4 Threads

For each of these cases, the amount of memory required for the table as well as the virtual address translation calculation is different, reducing the computation and/or the space requirement to its minimum. The size of the table on each thread is given by the number of blocks in the array except when the blocksize is 1, in which the size of the table is THREADS elements, as in Figure 8. In addition, the expensive modulo and divide operators present in address calculations when blocksize is finite, should be replaced when possible with bit-operators in high quality implementations.



Only first elements on each thread are saved … since all array data are contiguous

Figure 8a MMTB RT Mechanism with Default Blocked Shared Arrays

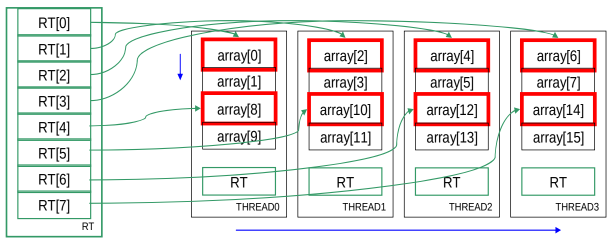
Consider shared [1] int array[16];

<u>To Initialize RT:</u>
$\forall$i $\in$ [0,THREADS-1],
RT[i] = _get_vaddr(array[i]);

<u>To Access array[ ]:</u>
$\forall$i $\in$ [0,15],
array[i] =_get_value_at(
RT[i mod THREADS] + ($\frac{i}{THREADS}$));

Figure 8b Example Initializing and Using the MMTB RT Mechanism with Default Blocked Shared Arrays

Figure 8 Example of MMTB with RT, Using Default Blocked Shared Array and 4 Threads



Only first elements of each block are saved … since all block data are contiguous

Figure 9a MMTB RT Mechanism with Arbitrary Blocked Shared Arrays

Consider shared [2] int array[16];

<u>To Initialize RT:</u>
$\forall$i $\in$ [0,7],
RT[i] = _get_val(array[i*blocksize(array)]);

<u>To Access array[ ]:</u>
$\forall$i $\in$ [0,15],
array[i] = _get_value_at(RT[$\frac{i}{blocksize(array)}$] +
(i mod blocksize(array)));

Figure 9b Example Initializing and Using the MMTB RT Mechanism with Arbitrary Blocked Shared Arrays

Figure 9 Example of MMTB with RT, Using Arbitrary Blocked Shared Array and 4 Threads

These two strategies, FT and RT, are creating additional dynamically allocated tables corresponding to each shared object during compilation time. The initialization of each FT or RT table, however, happens during the execution of the program, at the UPC runtime initialization. In case of shared pointers and other dynamically allocated objects, the RT or FT tables are updated at the same time as the shared object.

The RT MMTB mechanism is using less memory than the FT mechanism, by a factor equal to the size of the array blocks. However, where the FT strategy performs only one memory access to do the address translation, RT requires an additional address calculation step. On the other hand, the current compiler implementations seem to rely completely on pointer arithmetic, which is the source of the high access overhead. The measurements for a base UPC compiler implementation are estimated based on the prior analysis of the GCC-UPC compiler and are mainly focused on the address translation while ignoring any shared pointer arithmetic.

For a shared array of N elements with B as blocksize	Storage requirements per shared array (E: element size in bytes, P: pointer size in bytes)	# of memory accesses per shared memory access	# of arithmetic operations per shared memory access
UPC.O0	$N \cdot E$	At least 25	At least 5
UPC.O0.FT	$N \cdot E + N \cdot P \cdot THREADS$	1	0
UPC.O0.RT	$N \cdot E + \frac{N}{B} \cdot P \cdot THREADS$	1	Up to 3

Table 2 Storage, Memory Accesses and Computation Requirements of the Different Memory Model Translation Schemes

Table 2 provides a comparison among these three choices along with the associated trade-offs, with respect to storage requirements as well as memory operations and computations. The number of memory operations is proportional to the number of computational steps needed for the address calculation measurement. Thus, as far as the UPC.O0 is concerned, the measurements shown are reflecting only the get or put functions inside the UPC run-time environment. All the data in this table are obtained from the analysis of the source code of the current compiler implementations on the SGI [INT04].

5. Performance Impact of the Memory Model Translation Buffer Mechanism

5.1 STREAM BENCHMARK

To understand the impact of the proposed methods on long and short accesses, the STREAM benchmark has been run again but now using the optimizations FT and RT. Table 3 shows the results obtained, for both FT and RT, and compares them to the un-optimized C and UPC versions, as shown in Table 1.

First, it can be clearly seen that block operations (block copy, put and get) perform extremely well with and without any optimizations. This is because in these operations only the starting address of the block is needed. Thus, regardless of the cost of this single translation, it is amortized over the relatively large data.

The basic FT translation optimization, on the other hand, is giving a performance boost to operations that require only short accesses, such as array set or copy, sum and scale. Performance improvements over the non-optimized UPC are observed in the range of 2 to 4 folds. However, these improvements are still not performing as well as the private cases.

The second MMTB mechanism, RT, is performing even better than FT, and close to the level of performance of accesses to the private space and C. In most of the times it doubles the bandwidth given by the first mechanism, providing UPC with shared memory bandwidth that is as fast as C and hand-tuned UPC. One explanation for the good behavior of RT over FT is the extremely small RT table, which can be always kept in the cache, as the infinite blocksize is used in the STREAM benchmark.

		Bulk Operations			Element-by-Element Operations			
		Memory copy	Block Get	Block Put	Array Set	Array Copy	Sum	Scale
UPC Private		127	N/A	N/A	173	106	215	107
UPC + FT	UPC Local Shared	140	115	134	95	52	105	54
	UPC Shared (SMP)	131	115	134	94	52	104	54
	UPC Shared (Remote)	113	108	130	75	45	79	47
UPC + RT	UPC Local Shared	140	114	134	175	87	217	109
	UPC Shared (SMP)	131	114	134	175	87	217	109
	UPC Shared (Remote)	113	107	130	119	69	124	85

Table 3 STREAM Benchmark Revisited with FT and RT Enabled Compiler Optimizations (Bandwidths in MB/sec)

5.2 The GUPS BENCHMARK

Figure 10 shows the performance of the GUPS microkernel. The UPC version, which does not have any handtuning, is performing better than MPI. This is due to the fact that UPC is more efficient in short accesses than MPI.

In addition, it can be seen that the RT strategy is having a clear performance advantage. This is because the computation of the virtual address is simple since the default blocksize is used in GUPS for the base and indices arrays. In addition, the RT strategy address tables are only composed of THREADS elements, whereas FT uses tables of sizes similar to the original data structures, 1M and 4M elements. Thus, there is less competition between the RT tables and the GUPS data arrays. Due to the random accesses, cache misses increase as the number of CPU increases. Thus, the performance of all methods is similar performance when the number of processors increases.

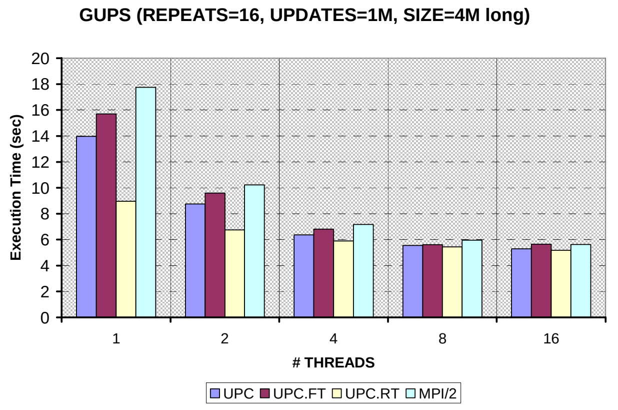


Figure 10 Performance of GUPS Using the new MMTB Strategies

5.3 SOBEL EDGE BENCHMARK

The performance of the Sobel Edge Detection is presented in Figure 11. The image size is set to 2048x2048. The execution time is almost 9 times faster after having privatized all local shared memory accesses and created a private copy of the remote data lines being accessed (prefetching optimization). The MPI performance is similar to the UPC hand-tuned version.

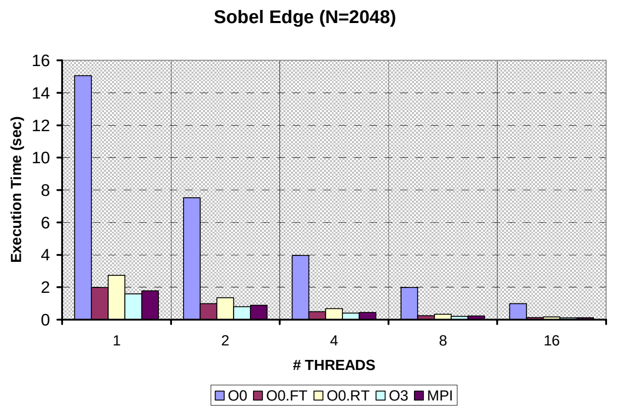


Figure 11 Performance of Sobel Edge Detection Using the New MMTB Strategies

The memory model translation buffer mechanisms FT and RT are delivering comparable performance to the fully hand-optimized version. In addition, FT and RT are performing around 6 to 8 folds the speed of the regular UPC version. The RT strategy is performing slower than FT in this case since the address calculations for the arbitrary block size case, as illustrated in Figure 9, are complex with three arithmetic operations per shared memory access. FT on the other hand is performing almost as good as the hand-tuned version.

5.4 MATRIX MULTIPLICATION

Figure 12 shows the performance of the matrix multiplication kernel using UPC and MPI with N=256. From the figure, it is clear that the hand-optimized UPC and MPI are performing identically. The RT MMTB strategy is performing roughly two times faster than the non-optimized UPC version, and two times slower than FT. On the other hand, the performance given by FT is close to that of the hand-tuned UPC and MPI codes.

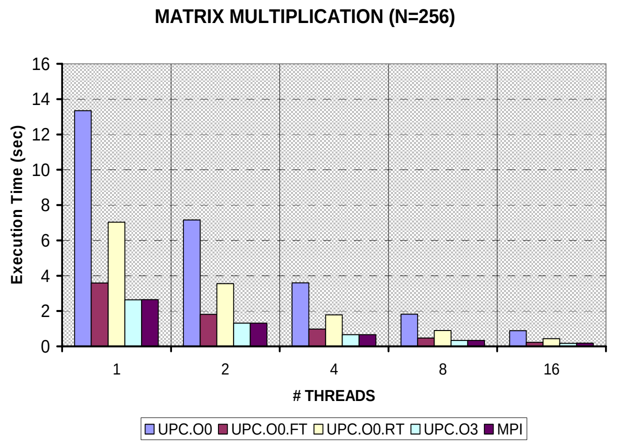


Figure 12 Performance of Matrix Multiplication Using the New MMTB Strategies

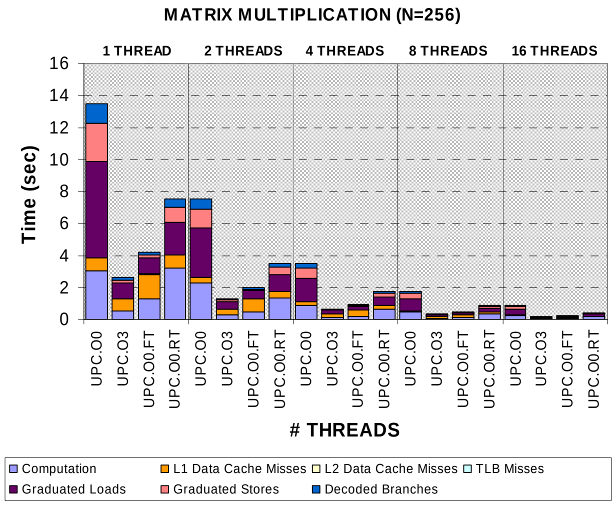


Figure 13 Hardware Profiling of the Different Variants of the Matrix Multiplication Workload

In an attempt to explain such significant differences in the FT and RT behaviors, a low-level comparative profiling study is presented in Figure 13. The study investigates the contributions of L1 and L2 data cache misses, TLB misses, graduated loads and stores, decoded branches and computations to the execution time. This study is conducted using the perfex monitoring tool and hardware performance counters available for the R10000 processors. The FT strategy is showing an increase of the L1 data cache misses, as compared to the others, which is clearly due to the large table size. RT is keeping L1 data cache misses at the same level as O0 and O3. However, RT is showing an increase in computations, as compared to FT and O3. This is due to the fact that RT is performing a lot of computations during the memory model address translation process. Since the shared matrices A, B and C are blocked using the largest chunk of rows or columns, and RT is using the arbitrary block size, a significant computational overhead is introduced.

	Shared array	Total Size in memory (in bytes)
UPC.O0	A,C	$N^2 \cdot sizeof(double)$
	B	$N^2 \cdot sizeof(double)$
UPC.O3	A,C	$N^2 \cdot sizeof(double) + THREADS \cdot sizeof(double*)$
	B	$N^2 \cdot sizeof(double) + THREADS \cdot N^2 \cdot sizeof(double)$
UPC.O0.FT	A,C	$N^2 \cdot sizeof(double) + THREADS \cdot N^2 \cdot sizeof(double *)$
	B	$N^2 \cdot sizeof(double) + THREADS \cdot N^2 \cdot sizeof(double *)$
UPC.O0.RT	A,C	$N^2 \cdot sizeof(double) + THREADS \cdot sizeof(double *)$
	B	$N^2 \cdot sizeof(double) + THREADS \cdot N \cdot sizeof(double *)$

Table 4 Memory Requirements for Each Variant for the Matrix Multiplication Workload

In order to analyze in more depth the memory requirements of each of the UPC variants, the matrix multiplication kernel is considered since it is a computational kernel that is largely used in scientific applications. The memory requirements of matrix multiplications are introduced in Table 4, where `sizeof`(x) means the size of x in bytes. From this table, the non-optimized UPC version is the variant which requires the lowest amount of memory. The hand-optimized version uses a private pointer to access A and C and a private copy of the complete B matrix to reduce remote accesses. On the other hand, the MMTB FT requires space for the shared matrices as well as complete private arrays of addresses of the same dimensions as A, B and C. RT increases the memory requirements over the non-optimized case by the space needed to hold a few private pointers only, which makes RT the second least demanding variant. Thus, RT is clearly a tradeoff between the memory space used and the computational effort.

5.5 NAS PARALLEL BENCHMARKS

Figure 14 to Figure 18 are showing the performance of the NAS kernels over Class C. Each UPC variant is represented, as well as the execution time of O0 using the MMTB compiler optimization. The MMTB mechanisms are consistently giving significant performance boost over O0. Such performance improvement is depending on how intensively shared variables are used.

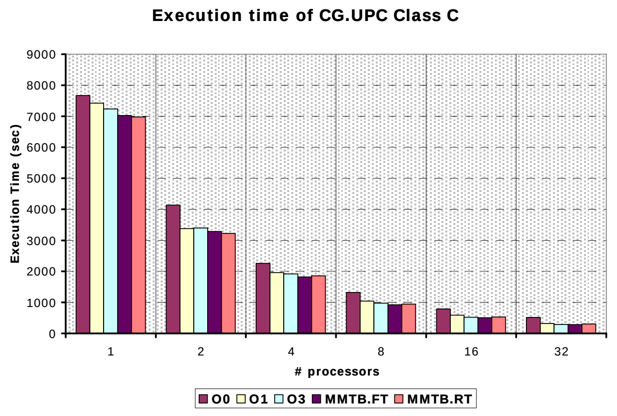


Figure 14 Performance of CG class C

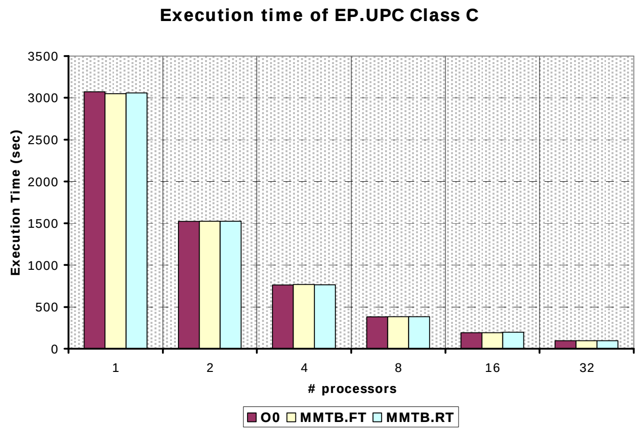


Figure 15 Performance of EP class C

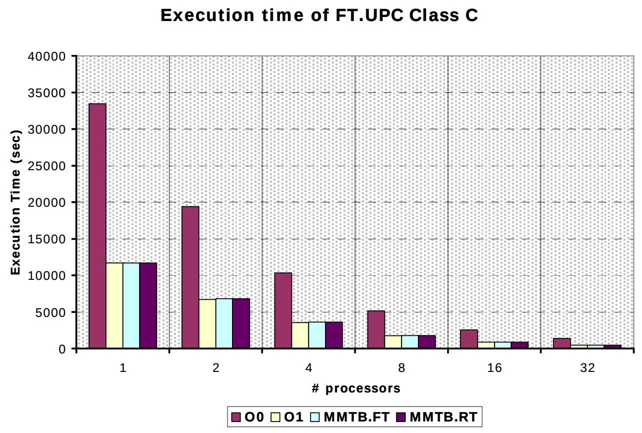


Figure 16 Performance of FT class C

In EP, Figure 15, very little shared data is used, thus the FT and RT variants are similar to O0 in performance. In Figure 14, the shared objects are used only during the communication phases in CG, which explains the small performance discrepancy between the O0, O3 and the MMTB mechanisms. Figure 16 through Figure 18, and especially Figure 18 with MG, are showing performance boosts from using FT and RT over the non-optimized O0 variant.

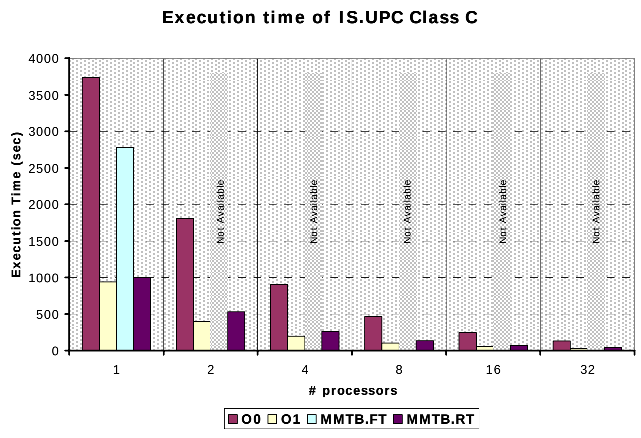


Figure 17 Performance of IS class C

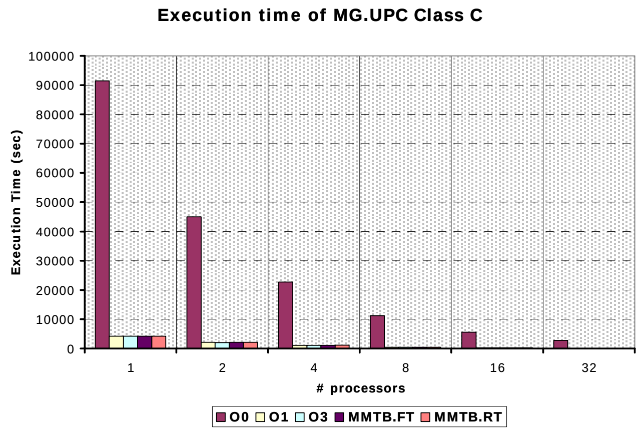


Figure 18 Performance of MG class C

MMTB mechanisms are delivering similar performance to their hand-tuned counterparts. Differences between the FT and RT variants are mostly present in IS, since the shared object is represented as a huge linear array, compared to the small arrays of structures used in the other workloads. IS.MMTB.FT increases the memory requirements with the number of threads, which leads to memory allocation errors with number of THREADS higher than 2. The corresponding MMTB.RT variant is making good use of the default blocksize of the array. It maintains memory requirements of the same level as the non-optimized code while having a performance similar to the optimized code.

6. Conclusions

In this paper, two new strategies that can help automatically reducing the memory model translation overhead in distributed shared memory programming paradigms were proposed and prototyped. The prototypes were created using the GCC-UPC compiler. These strategies attempt to achieve the best performance possible, while maintaining the ease-of-use of the UPC language.

The first strategy, referred to as FT, is based on a full-look-up table of virtual addresses. This table has a size equal to that of the shared object. While it is roughly doubling the memory requirements of applications, it allows a direct-mapping between the UPC shared user space and the architecture shared virtual space. However in case of workloads with large datasets, the tables created by FT compete with the application for resources, such as L1 and L2 data caches and even main memory, which may cause a performance drop.

The second strategy, RT, is an improved version of FT. It reduces the memory usage while trying to keep address calculation simple. The memory requirements as well as the calculation method used for the address translation process in this scheme depend on mainly the blocksize of the shared array. RT is a tradeoff between the large space needed by the FT mechanism and the address translation computational complexity of current implementations.

It has been shown that FT and RT are producing a similar level of performance to that of the hand-optimized O3 version of UPC and MPI. Recall for example the cases of Sobel Edge Detection and GUPS. Substantial cache misses arise from the FT mechanism, as seen in the Matrix Multiplication kernel with an increase of the L1 date cache misses. However, RT causes an increase in computations.

With these FT and RT strategies, as observed with the NAS kernels, can one achieve automatically a speedup of up to a factor of 8 as compared to the non-optimized version. This can deliver a level of performance similar to that of the hand-tuned version, and thus to MPI, while maintaining the ease-of-programming of UPC.

References

[BRO95] Brooks, Eugene and Warren Karen, *Development and Evaluation of an Efficient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared-memory Multi-processor and Distributed-memory massively Parallel Architectures*, Poster SuperComputing 1995, San Diego, CA, December 3-8, 1995.

[CAN03] Cantonnet François, Yao Yiyi, Annareddy Smita, Mohamed Ahmed, El-Ghazawi Tarek, *Performance Monitoring and Evaluation of a UPC Implementation on a NUMA architecture*, International Parallel and Distributed Processing Symposium (IPDPS), Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO) workshop, 2003, Nice France

[CAR99] Carlson William and Draper Jesse, *Distributed Data Access in AC*, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara, CA, July 19-21, 1995, pp.39-47

[CUL93] Culler, Dusseau Andrea, Goldstein Seth Copen, Krishnamurthy Arvind, Lumetta Steven, Von Eicken Thorsten and Yelick Katherine, *Parallel Programming in Split-C*, Proceedings of SuperComputing 1993, Portland, OR, November 15-19, 1993

[ELG01a] El-Ghazawi Tarek, *Programming in UPC*, Tutorial (http://upc.gwu.edu), April 2001

[ELG01b] El-Ghazawi Tarek and Chauvin Sébastien, *UPC Benchmarking Issues*, 30th Annual Conference IEEE International Conference on Parallel Processing,2001 (ICPP01) Pages: 365-372

[ELG02] El-Ghazawi Tarek and Cantonnet François, *UPC Performance and Potential: A NPB Experimental Study*, SuperComputing 2002, IEEE, Baltimore MD, November 2002

[ELG03] El-Ghazawi Tarek, Carlson William and Draper Jesse, *UPC Language Specifications v1.1* (http://upc.gwu.edu), October 2003

[GAE02] Gaeke Brian and Yelick Katherine, *GUPS (Giga-Updates per Second) Benchmark*, Berkeley, 2002

[INT04] Intrepid, The GCC UPC Compiler for SGI Origin Family v3.2.3.5 (http:// www.intrepid.com/upc/)

[ISO99] ISO/IEC 9899:1999, *Programming languages — C*, December 1999.

[MCC95] McCalpin John, *Sustainable memory bandwidth in current high performance computers*, Technical report, Advanced Systems Division, SGI., October 12, 1995

[NPB02] *NAS Parallel Benchmark Suite*, NASA Advanced Supercomputing, 2002, http://www.nas.nasa.gov/Software/NPB