Address translation optimization for Unified Parallel C multi-dimensional arrays

Olivier Serres, Ahmad Anbar, Saumil G. Merchant, Abdullah Kayi and Tarek El-Ghazawi
NSF Center for High-Performance Reconfigurable Computing (CHREC),
Department of Electrical and Computer Engineering,
The George Washington University
{*serres, anbar, apokayi*}*@gwmail.gwu.edu,* {*smerchan, tarek*}*@gwu.edu*

***Abstract*—**
Partitioned Global Address Space (PGAS) languages offer significant programmability advantages with its global memory view abstraction, one-sided communication constructs and data locality awareness. These attributes place PGAS languages at the forefront of possible solutions to the exploding programming complexity in the many-core architectures. To enable the shared address space abstraction, PGAS languages use an address translation mechanism while accessing shared memory to convert shared addresses to physical addresses. This mechanism is already expensive in terms of performance in distributed memory environments, but it becomes a major bottleneck in machines with shared memory support where the access latencies are significantly lower. Multi- and many-core processors exhibit even lower latencies for shared data due to on-chip cache space utilization. Thus, efficient handling of address translation becomes even more crucial as this overhead may easily become the dominant factor in the overall data access time for such architectures. To alleviate address translation overhead, this paper introduces a new mechanism targeting multi-dimensional arrays used in most scientific and image processing applications. Relative costs and the implementation details for UPC are evaluated with different workloads (matrix multiplication, Random Access benchmark and Sobel edge detection) on two different platforms: a many-core system, the TILE64 (a 64 core processor) and a dual-socket, quad-core Intel Nehalem system (up to 16 threads). Our optimization provides substantial performance improvements, up to 40x. In addition, the proposed mechanism can easily be integrated into compilers abstracting it from the programmers. Accordingly, this improves UPC productivity as it will reduce manual optimization efforts required to minimize the address translation overhead.

I. Introduction

The continuing quest for higher performance has initiated a new focus, emphasizing thread- and task-level parallelism, leading to the emergence of homogeneous multi- and many-core processors. The shift towards using multiple cores on the same die relies on thread-level parallelism (TLP) and accordingly parallel programming to achieve performance improvements. This programming paradigm shift puts the responsibility of achieving higher performance on the shoulders of the programmer. The programmer must now understand and exploit parallelism explicitly by utilizing some kind of parallel programming environment. To exploit the full potential of these new multi-core processors, new design methodologies and languages are needed that can abstract low-level details, but at the same time enable the mainstream programmer to extract fine-grained parallelism. HPC community has, over the last few decades, extensively researched parallel programming languages and methods, results and lessons from which form a strong foundation to develop new methodologies amenable to the mainstream programming community. A relatively newer design paradigm that has influenced many new HPC languages is the Partitioned Global Address Space (PGAS) model. PGAS offers a global, logically shared memory space for all threads, with locality awareness and one-sided communication constructs.

It offers advantages on two fronts: (i) performance, and (ii) ease of use, significantly enhancing user productivity. Data-locality awareness resulting from partitioned address space and lower communication overheads offer high performance. Global shared memory logical view and one-sided communication constructs facilitate ease of use. Furthermore, PGAS languages are nearly ubiquitous, which helps for code portability. Several programming languages such as Unified Parallel C (UPC), Co-Array Fortran (CAF) and Chapel follow the PGAS parallel programming paradigm.

The partitioned shared memory view adopted by UPC, as other PGAS languages, uses an address translation mechanism while accessing shared memory to convert shared addresses to local addresses. Already costly in terms of performance in distributed memory systems, the address translation mechanism becomes the main bottleneck in machines with shared memory support where the data access latencies are significantly lower. Multi- and many-core processors exhibit even lower latencies for shared data due to on-chip cache space utilization. Earlier studies showed significant performance issues that arise from mis-handling of cache hierarchies in multi-core based systems [1]. Thus, efficient handling of address translation becomes even more crucial as this overhead may easily become the dominant factor in the overall access time for such architectures.

In this paper, we present a novel optimization technique to minimize the address translation overhead in the most common case of multi-dimensional arrays. Multi-dimensional arrays are abundantly used in most scientific and image processing applications. In addition, our proposed optimizations can easily be integrated into compilers. This will further improve the programmability of UPC as it will eliminate

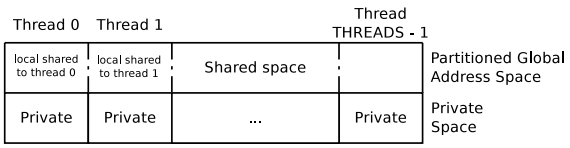


Figure 1: The UPC memory model

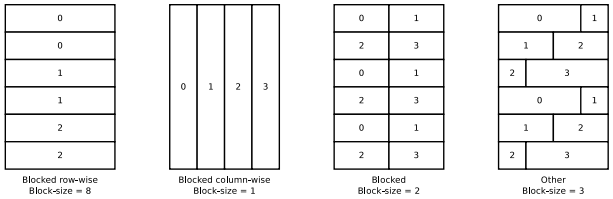


Figure 2: Different layouts for a UPC shared array (6*4) distributed among 4 threads. The number in each block represent the thread affinity of the given block.

the manual optimization efforts required to offset the address translation overhead. Furthermore, different implementation alternatives of our proposed optimization mechanism are studied. In addition, empirical results on two different platforms making a strong case for an improved address translation are presented.

The rest of this paper is organized as follows. UPC and its shared memory model is presented in Section II. In Section III, previous approaches to reduce the address translation overhead are discussed. Our proposed optimization is described in detail in IV including cost analysis and implementation trade-offs. Section V presents the benchmarks used and the corresponding results. Finally, Section VI concludes the paper.

II. Presentation of UPC and its shared memory model

UPC is an explicit parallel extension of the C language supporting the PGAS programming paradigm [2], [3]. It offers a C language syntax. This eases the learning curve and provides a high productivity development environment [4]. The UPC execution model supports SPMD style parallel programming where a specified number of threads execute cooperatively in parallel on multiple processors. It provides various synchronization mechanisms for the threads such as barriers, locks, and memory consistency control statements.

The UPC memory model supports and extends the PGAS memory model concepts. As in PGAS, the UPC memory model provides a global shared memory space partitioned to provide affinity for portions of the shared address space resident locally. This provides a global view to the user, effectively improving productivity [5]. In addition, each thread is also given a private space that is not accessible by other threads. Variable declarations can explicitly state shared versus private storage space. A pointer-to-shared

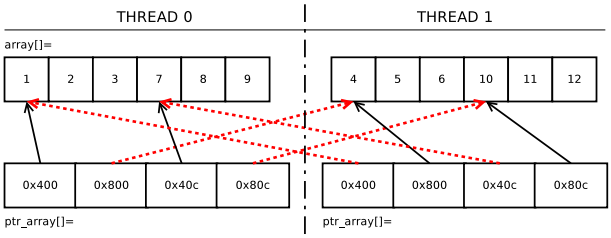


Figure 3: Illustration of the memory distribution of `array` and its associated row pointer array `ptr_array`. Dashed arrows represent inter-thread accesses.

variable declaration can reference all locations in the shared space. A private pointer can only reference the addresses in the thread private address space or its local portion of the shared space. Syntactically, both remote and private memory accesses are simple variable assignment statements, providing a standard logical abstraction for the one-sided remote communications. Figure 1 shows the UPC memory model.

In UPC, a blocking factor (or block-size) is utilized to control the shared array distribution among the threads. If a blocking factor of size n is specified, the array is distributed by block of n elements. The blocking factor can only be represented by a scalar, so it is not possible in UPC to have more complex distributions, for example 2D block-cyclic distribution [6] is not possible. Shared pointers can also be defined with a blocking factor. The blocking factor size of the pointer has to match the block-size of the array it is pointing to in order to traverse the array in the correct order.

The address translation process consists of determining the actual physical location of the data in the shared space. First the location of the thread that owns the data is determined, then the virtual address inside this thread address space is computed.

III. Related work

The address translation overhead for remote accesses in UPC, when not manually optimized, is important resulting in programs that can be significantly slower in performance as compared to their C counterparts. Experienced users typically use private pointers to close this performance gap and manually optimize their codes. In such cases, particular care has to be taken for the boundary conditions. Unfortunately, this impedes the programmability advantages of UPC for novice users due to the associated complexity and also makes the code difficult to read.

According to the latest UPC specifications [2], private pointers can only access the shared memory part of the local thread. This greatly reduces the optimization possibilities available to the users. Users cannot use private pointers to access data of other threads even if those threads are using the same physical memory. Only recently, HP UPC

group proposed to extend the specifications with an API allowing to cast a shared pointer to a private pointer even though the shared pointer does not point to the local space of the thread [7]. This will only be possible for the physical memory accessible by the thread.

Due to the difficulty for the user to manually optimize the code, it is essential to provide fast address translation mechanisms. Some previous research studies targeted the performance issues resulting from address translation overhead. In [8], Cantonnet *et al.* took the approach of pre-computing tables with address-translations. The address translation code is reduced to the cost of a table look-up. Unfortunately, storing the complete table is often unfeasible due to its memory requirement. This is solved by introducing reduced-tables which only includes the translation for the first element of each block. Additional arithmetic operations are required to translate other addresses, trading some computation time for memory. Multi-dimensional arrays were not considered in their work.

In [9], the authors highlight the inefficiencies of the UPC address translation. The major bottleneck for performance identified by the authors is the address translation and the associated complex arithmetic overheads involved. To remedy this problem, the authors introduce the block-major layout. In the block-major layout the arrays, on a physical shared memory system, are stored contiguously as in C regardless of their UPC block-size. This makes the cost associated with shared references to the array to be comparable to that in C. The limitation of that method is that it is not possible anymore to derive a private pointer from the optimized shared pointers (due to the incompatible arrays layout). The optimization is automatically disabled for arrays using private pointers.

To mitigate the address translation overhead, the Berkeley UPC compiler performs two specific optimizations [10]. First, it uses "phaseless" pointers or pointers with a zero phase. Since the phase is always zero, it doesn't need to be included in the address translation calculations. This happens in two cases; (i) when the block-size is 1, and (ii) when the block-size is infinity (represented as a block-size of 0). In the second optimization the Berkeley compiler offers an 8 byte packed representation which has lower overhead when compared to a "struct" representation. On the other hand, the IBM XLUPC compiler and runtime system uses a shared variable directory (SVD) to share the location of shared variables. The runtime system employs a local cache to reduce SVD accesses and allow RDMA accesses [11]. This is designed for large scale system and does not particularly address multi- and many-core systems that have lower latency.

IV. Proposed optimization

The following UPC array declaration:

`shared [`b`] int array[`d_n`][`d_{n-1}`]...[`d_1`];`

declares a shared array of type int, of dimension n, with a block-size b. When $n > 1$, the array is a multidimensional array.

The optimization targets arrays that are distributed row by row among the UPC threads (see Figure 2) . This is the case when block-size is a multiple of the number of elements in a row of the array; i.e. when $d_1 \equiv 0 \pmod{b}$. This type of arrays are commonly found in scientific and image processing applications [12].

The address translation for this type of arrays can be greatly optimized by generating extra arrays of pointers to access those arrays.

Let's consider the following array:

```
1 shared [3] int array[4][3] = { 1,  2,  3,
                                 4,  5,  6,
                                 7,  8,  9,
                                10, 11, 12};
```

The block-size of this array is 3, so it will be distributed, in a round-robin fashion, by block of 3 elements to the UPC threads. If we have two threads, the distribution will be as in Figure 3. Also, the elements $1, 2, 3, 7, 8, 9$ will be stored continuously in the local shared space of thread 0 and $4, 5, 6, 10, 11, 12$ will be in the space of thread 1.

If the thread private spaces are directly accessible from the other threads (as when the UPC threads are implemented with pthreads, for example), it is possible to construct an array of pointer pointing to the beginning of each row: `ptr_array` in Figure 3.

Referencing an element using the array itself or `ptr_array` is equivalent, for example:

```
1     assert( array[2][1] == ptr_array[2][1] );
```

The advantage of using `ptr_array` is that it replaces the expensive address translation with an array look-up. The generation of `ptr_array` and the substitution of `array` by `ptr_array` can easily be done by the compiler.

The optimization can also be extended to arrays blocked by column; in that case, the array would be transposed by the compiler, effectively transforming the problem to a row blocked array. Some C and other compilers already transpose arrays in order to ensure that the accesses are aligned with the cache lines.

A. Locality of the pointer arrays

For better performance, each thread should have the lowest latency when accessing its local space of the shared array; this is why the generated pointer arrays locality should as much as possible match the locality of the targeted array. For example, if a row is local to a specific UPC thread, the pointer to this row should also be local to the same UPC thread, as it is very likely that this row will be mostly accessed by that specific thread. To provide this locality, it is important that the pointer array is arranged correctly in memory.

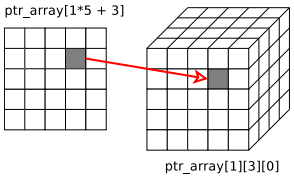


Figure 4: Two steps lookup implementation

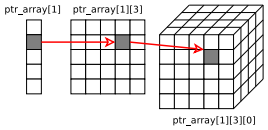


Figure 5: $n-1$ steps lookup implementation (2 steps in this case)

Without duplicating this array, it is not always possible to ensure that the locality of the pointers match the locality of the array. Mainly, this is due to the fact that the memory is managed by the system with unit size of a memory page. A memory page (usually 4KB on x86 systems) tends to be big compared to the pointer array size. For the Nehalem system (see Figure 7), in order to obtain the same latency from all the cores, the pointer arrays should be duplicated twice, once for each memory controller. On the TILE64 system, the pointer arrays have been duplicated once for each thread, so that for each memory page a home cache is set and the data of that page will only be cached in this specific cache. To further reduce the memory usage, it is possible to mark some pages as read only so that they can be cached anywhere on the chip. The optimization cost in terms of memory is analyzed in the next subsection.

B. Optimization cost

There are two alternatives to implement this optimization, both shown in Figures 4 and 5 for 3D arrays. Figure 4 shows a one level lookup, where the row pointers are stored in an array which is one dimension less than the shared data array. Figure 5 shows the $n-1$ levels lookups, where an array in one level is one dimension less than and contains pointers to first elements in its rows of the array in the array in the level that follows it. In terms of computation power, to find the final address of an array element, the first alternative will need an address calculation based on $n-1$ index which involves both multiplication and addition. The second alternative will need $n-1$ table lookups. Dynamic allocation of multi-dimensional arrays (through a library or new language constructs) can also be implemented by using $n-1$ table lookups. In modern processors, a special instruction is present to perform this type of operation, making this a fast operation. Also, it is important to note that, in most of the cases, some lookups can be optimized as in the following code example. The table lookups to obtain the beginning of each row (`ppb_j`) are done outside of the main loop. The main loop only has index computations to perform and no table lookup.

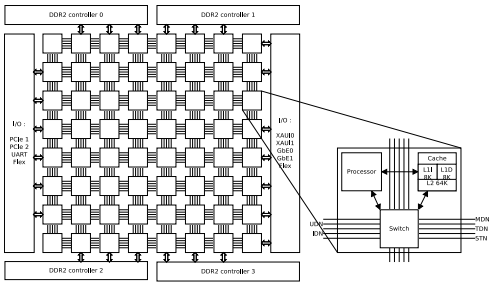


Figure 6: TILE64 architecture diagram [13]

```
for( i=nb_lines*MYTHREAD; i<nb_lines*(MYTHREAD+1); i++ ){
    int *ppc_i = pc[i];
    for( j=0; j<MATRIX_SIZE; j++ ){
        int r = 0, *ppb_j = pb[j];
        for( k=0; k<MATRIX_SIZE; k++ )
            r += pa[i * MATRIX_SIZE + k] * ppb_j[k];
        ppc_i[j] = r;
    }
}
```

1) Shared pointer arrays: The minimum memory cost to build the pointer arrays c_{m1} is as follows:

$$c_{m1} = \sum_{j=2}^{n}(\prod_{i=j}^{n} d_i * \texttt{sizeof(void*)}) \tag{1}$$

$$= d_n*(1+d_{n-1}*(1+d_{n-2}*(...+d_2)))*\texttt{sizeof(void*)} \tag{2}$$

This is obtained by only having one copy for the whole program. It can provide a good locality for system with a common shared cache; e.g. a single socket Nehalem system.

2) Duplication of the pointer arrays: The maximum memory cost for this optimization is obtained when the pointer tables are duplicated on each thread ($c_{mmax} = t * c_{m1}$). This is the option that we have chosen for the TILE64 processor, in order to have a correct locality.

3) Case of arrays with $d \geq 3$: Based on equation 1, it is possible to reduce the memory used by the pointer arrays. This is done by sorting the dimension of the array from the smallest to the biggest. Considering the following array:

```
shared [4] int array[16][8][4];
```

The pointer arrays can be built as if the array has the following dimensions:

```
shared [4] int array[8][16][4];
```

Based on equation 1, the pointer arrays should occupy $16*(8+1)*8 = 1152$ bytes. By sorting the array indices, the look-up tables will only use $8*(16+1)*8 = 1088$ bytes. For this particular example, the reduction is limited; but for array of higher dimensions, such a reduction will be more important.

Table I: UPC overhead over C code

	TILE64						NEHALEM PLATFORM					
	Matrix mult.		Sobel edge		Random access		Matrix mult.		Sobel edge		Random access	
UPC Optimization	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
CC -O3 (time s)	12.40		0.76		36.57		0.43		0.0015		2.56	
UPC 1 thread (time s)	649.12	16.35	6.81	0.78	88.37	39.90	9.91	1.37	0.104	0.0016	3.65	2.72
UPC overhead over C	52 x	32 %	9 x	3 %	142 %	9 %	23 x	3 x	69 x	1 %	43 %	6 %

Table II: Optimization memory cost. *The overall optimization cost range is shown, depending on the number of running threads.*

	TILE64			NEHALEM PLATFORM		
	Matrix mult.	Sobel edge	Random access	Matrix mult.	Sobel edge	Random access
Data size	$1024^2 * 4 * 3$	$1024^2 * 2$	$1024^2 * 4$	$1024^2 * 4 * 3$	$1024^2 * 2$	$1024^2 * 4$
Pointer array size	$1024 * 4 * 3$	$1024 * 4 * 2$	$1024 * 4$	$1024 * 8 * 3$	$1024 * 8 * 2$	$1024 * 8$
Optimization cost	$0.1 - 3.1\%$	$0.4 - 12.5\%$	$0.1 - 3.1$ %	$0.2 - 0.4$ %	$0.8 - 1.6$ %	$0.2 - 0.4$ %

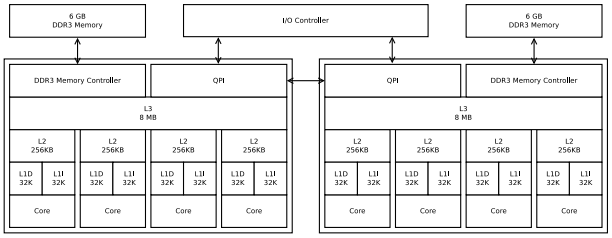


Figure 7: Architecture of our Nehalem system

V. EXPERIMENTAL SETUP AND RESULTS

A. Platforms

We tested our proposed optimization on two different platforms:

- **TILE64 system:** A TILE64 board comes with 4 GB of memory. The TILE64 processor features 64 identical 32-bits processor cores (tiles) interconnected with Tilera's iMesh on-chip network [14]. Each tile consists of a complete, full-featured 3-way VLIW processor as well as a 8KB of L1 data cache, 8 KB of L1 instruction cache, a 64 KB private L2 cache and a non-blocking switch that connect the tiles into the mesh. Due to the small sizes of the caches, the data locality awareness provided by PGAS languages is a crucial feature for performance. Four on chip memory controllers connect the tiles to on-board DDR2 memories. Figure 6 shows the architecture diagram of the TILE64 processor. It is also important to note, that some tiles are reserved to control on-board devices (PCIe bus, 10Gb Ethernet, ...). Thus, results from TILE64 system are reported at most for 32 cores.
- **Intel Nehalem system:** The Nehalem system is a dual Intel Xeon E5520 processor (quad core, 2.27 GHz) with 12 GB of memory (at 1066 MHz). Figure 7 details the cache hierarchy. Hyper-threading is enabled, allowing a total of 16 running threads.

The Berkeley UPC compiler version 2.8.0 with the pthreads conduit was used to compile the code on both platforms. The TILE64 platform is not a supported platform, so we had to cross-compile and patch GASNet [15] and the UPC runtime for Tilera's C compiler (tile-cc version 2.0.1.78377). It should be considered as an experimental platform. On the other hand, the Nehalem system is a more mature platform which has a wide-spread adoption; a correct level of address translation optimization is expected from the compiler. The C compiler used in the Nehalem system is the GNU C compiler (gcc version 4.4.3).

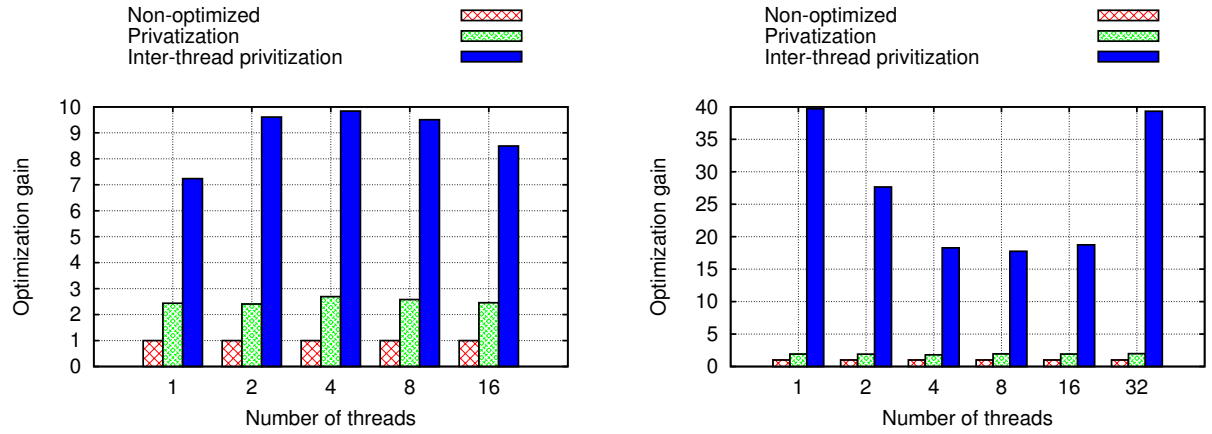
B. Benchmarks

We used three different benchmarks to test our approach:

- **Matrix multiplication :** Matrix multiplication is the building block of many scientific applications. It computes $A = B * C$; A, B and C being integer matrices of dimension 1024 * 1024. The ordinary row by column algorithm is used (complexity $O(n^3)$). The non-optimized matrix multiplication kernel is as follows:

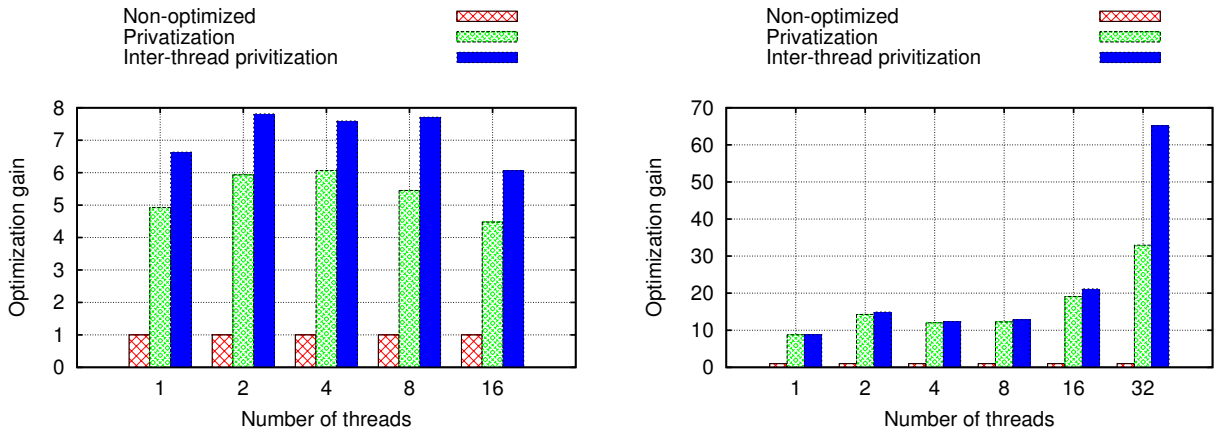
```
upc_forall( i=0; i<MATRIX_SIZE; i++; i/nb_lines ){
    for( j=0; j<MATRIX_SIZE; j++ ){
        int r = 0;
4       for( k=0; k<MATRIX_SIZE; k++ )
            r += a[i][k] * b_trans[j][k];
        c[i][j] = r;
    }
}
```

  It can be noted that the B matrix has been transposed in order to align the accesses with the cache lines. The privatization optimization is applied by making use of a private pointer to access the matrix A. The inter-thread privatization optimization is obtained by applying our optimization to all the matrices A, B and C; this totally replaces all the address translations by table look-ups.
- **Sobel edge :** The Sobel operator is a well known operator that is used in digital image processing for edge



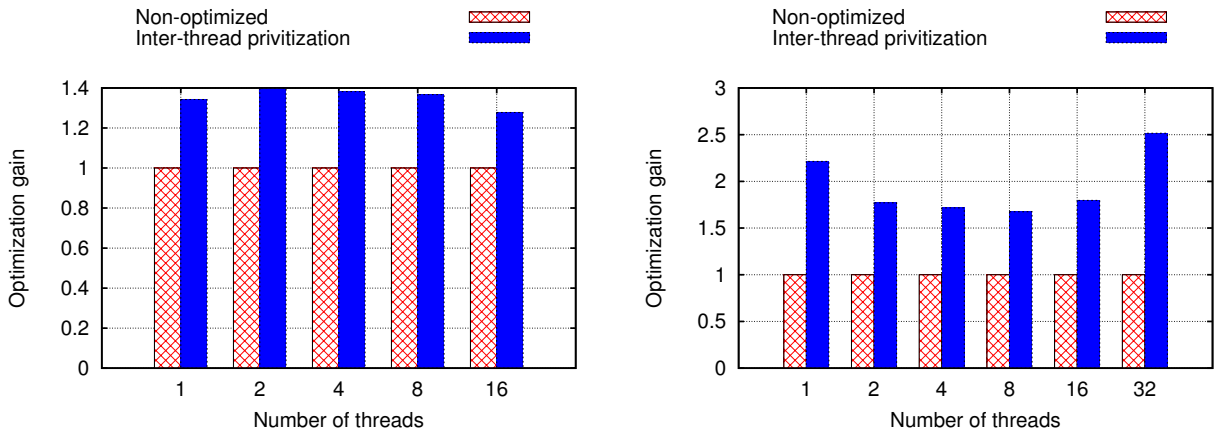
(a) Nehalem system (b) TILE64 system

Figure 8: Matrix multiplication, optimization gain for different levels of optimization



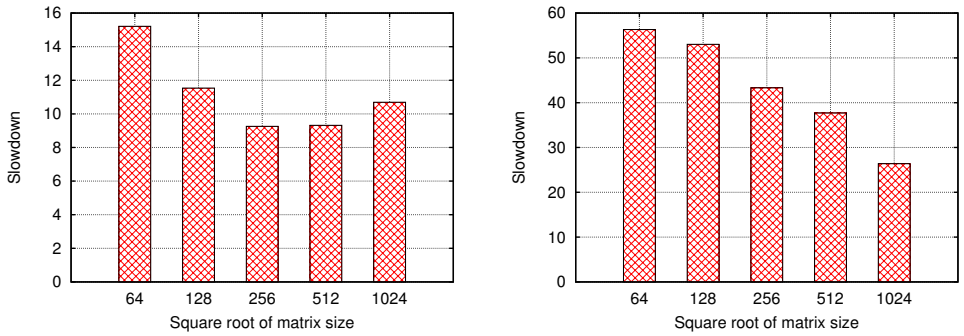
(a) Nehalem system (b) TILE64 system

Figure 9: Sobel edge image processing, optimization gain for different levels of optimization



(a) Nehalem system (b) TILE64 system

Figure 10: Random access benchmark, optimization gain for different levels of optimization



(a) On the Nehalem platform (b) On the TILE64 platform

Figure 11: Dataset size sensitivity - Comparison of the sequential C version with single thread UPC

detection [16]. Mathematically, the operator convolves two 3x3 kernels with an input 2D image to obtain an approximation of the derivates of the input image.
The input image is distributed to the threads such that each thread has a horizontal block assigned to it. Each thread works on its own assigned block and produces an equivalent part of the output image. By this distribution, each thread has the locality to all the needed data except for two rows of the input image. In the privatized version, a private pointer is used except for the first and the last row for which a shared pointer is used.

- **Random access :** This benchmark is very similar to the DARPA High Productivity Computing Systems (HPCS) Random Access benchmark [17]. This benchmark stresses the memory subsystem by doing random accesses. This is a very interesting test case for our optimization: it is not possible to predict where the next access will be. Thus, the full address translation has to be performed for each access. Typical optimizations are relatively difficult to implement.

C. Methodology and Results

The benchmarks were run with different number of threads on both platforms. Each run was conducted at least 5 times and the median was taken in order to reduce system noise. Compiler optimizations were enabled for all runs; only the level of manual optimization varies as stated in the related figures. Privatization refers to an optimization in which shared pointers are converted to normal C pointers for the local part of the shared array. Inter-thread privatization refers to our proposed optimization mechanism for address translation.

Table I shows the overhead of using UPC over C. This is obtained by comparing the execution of sequential C code with both the non-optimized and the inter-thread privatized UPC codes running with only one thread. Even though one would expect to get similar performance between C code and a UPC code with one thread, a dramatic slowdown (up to 69x slower) is observed for the non-optimized version, plain UPC code without manual optimizations. Once optimized with inter-thread privatization, i.e. once the address translation operation has been replaced by table look-ups, single thread UPC results become comparable to sequential C results. As it is expressed earlier, we observed a higher address-translation overhead for the workloads that better utilize the on-chip caches which have relatively lower access latencies. As such, random access benchmark results exhibit a lower overhead compared to Sobel edge and matrix multiplication. This is also illustrated in Figure 11 in which the overall UPC overhead with respect to C is plotted against the varying workload sizes. The address translation is known to be the dominant factor that causes this overhead.

Figure 8 through 10 show the results obtained for different number of UPC threads in our testbeds while running the aforementioned workloads. Corresponding extra memory costs from our optimized mechanism are reported for each workload and platform in Table II. All optimization gain results are normalized to non-optimized UPC runs for the same number of threads. Optimization gains are reported along with the raw execution timing results to clarify any issues that may arise because of the clock speed differences between two platforms. The optimization cost in terms of memory is low: in most of the cases it is inferior to 1%; the cache pollution being proportional to the memory cost is also being extremely low.

Matrix multiplication results can be found in Figure 8. Results present up to 10x and 40x improvements in Nehalem and TILE64 systems respectively. Since this kernel exhibits a higher cache utilization, our optimization would be able to provide large improvements. This supports our initial statement that reflects the importance of address-translation optimizations on multi- and many-core based systems which tend to rely more on cache utilization for performance. The Sobel edge benchmark has a working set larger than the

total of last level caches in both platforms. However, it still presents a repetitive data access scheme which can be better exploited by temporal and spatial locality. Corresponding Sobel edge benchmark results can be found in Figure 9. The privatization helps relatively a lot on this benchmark, but our optimization further improves the performance over the regular privatization optimization.

Random access benchmark presents a different scenario in terms of memory subsystem utilization by thrashing the cache and stressing only on memory accesses which are all random. Thus, improvements through address-translation optimizations are relatively limited for this benchmark. However, as shown in Figure 10 we still manage to get $\sim$ 2x performance improvements on both platforms.

It is also important to note that the UPC single thread overhead over C counterpart for the random access benchmark is fairly low (less than 10%). This can be counter intuitive at first as the number of loads is doubled due to the lookup tables' accesses. However, it should be noted that the table will be fully cached and its accesses will cost the fraction of an external memory fetch (on Tile 64: an L1-cache hit takes 2 clock cycles whereas a DDR2 memory fetch on an open page takes around 69 cycles).

VI. Conclusions and Future Work

The transition to multi- and many-core processors provides new ways of reaching performance gains mainly through Thread-Level Parallelism (TLP). PGAS programming languages are a good candidate to solve the associated programming productivity issues; unfortunately this comes with some additional overheads. Mainly, address translation overhead has shown to be a major bottleneck.

In this work, we demonstrated the importance of this bottleneck for multi- and many-core systems and presented a new mechanism to minimize this overhead for multi-dimensional arrays. Experiments were conducted on a dual-socket, quad-core Intel Nehalem system and a TILE64 system. Results from three different workloads have shown promising performance improvements up to 40x for matrix multiplication kernel. In addition, we have shown that our optimization helps the scalability of UPC applications. Furthermore, our proposed mechanism can easily be integrated into compilers which would reduce manual optimization efforts.

As a future work, we are planning to study the effects of this optimization in a distributed memory environment (clusters); reducing the address translation overhead for low latency networks like Infiniband is expected to improve UPC efficiency.

Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

References

[1] A. Kayi, E. Kornkven, T. A. El-Ghazawi, and G. Newby, "Application performance tuning for clusters with ccnuma nodes." in *11th IEEE International Conference on Computational Science and Engineering*, 2008, pp. 245–252.

[2] The UPC Consortium, "UPC language specifications v1.2 (`www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`)," 2005.

[3] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, May 2005.

[4] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the UPC language," in *Proceeding of the 18th International Parallel and Distributed Processing Symposium.*, April 2004, pp. 254–.

[5] R. Nishtala, G. Almási, and C. Caşcaval, "Performance without pain = productivity: data layout and collective communication in UPC," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 99–110.

[6] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 5–5.

[7] B. Wibecan, "Specification extension proposal: Privatization functions for UPC," in *UPC workshop at PGAS '09: the Third Conference on Partitioned Global Address Space Programing Models*, 2009. [Online]. Available: http://www2.hpcl.gwu.edu/pgas09/HP_UPC_Proposal.pdf

[8] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast address translation techniques for distributed shared memory compilers," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, 2005.

[9] W. Zhao and Z. Wang, "ScaleUPC: a UPC compiler for multi-core systems," in *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. New York, NY, USA: ACM, 2009, pp. 1–8.

[10] W.-Y. Chen, D. Bonachea, and K. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *Conference proceedings: 2003 International Conference on Supercomputing: June 23-26, 2003, San Francisco, California, USA*, vol. 4. Association for Computing Machinery, 2003, p. 63.

[11] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes, "Scalable RDMA performance in PGAS languages," in *IEEE International Symposium on Parallel Distributed Processing 2009*, May 2009, pp. 1 –12.

[12] A. Salah, O. Serres, J. Gaber, R. Outbib, and H. El-Sayed, "Simulation of the fuel cell thermal behavior with Unified Parallel C," Nov. 2007, pp. 149–152.

[13] Tilera Corporation, "`www.tilera.com`."

[14] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the Tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15–31, Sept.-Oct. 2007.

[15] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[16] I.Sobel and G. Fledman, "A 3x3 isotropic gradient operator for image processing," presented at a talk at the Stanford Artificial Project in 1968, unpublished but often cited.

[17] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger, "HPCC RandomAccess benchmark for next generation supercomputers," in *IEEE International Symposium on Parallel Distributed Processing - IPDPS 2009*, 23-29 2009, pp. 1–11.