Principles of Memory-Centric Programming for High Performance Computing*

Yonghong Yan
University of South Carolina
Columbia, SC
yanyh@cse.sc.edu

Ron Brightwell
Sandia National Laboratories
Albuquerque, NM
rbbrigh@sandia.gov

Xian-He Sun
Illinois Institute of Technology
Chicago, IL
sun@iit.edu

ABSTRACT

The memory wall challenge – the growing disparity between CPU speed and memory speed – has been one of the most critical and long-standing challenges in computing. For high performance computing, programming to achieve efficient execution of parallel applications often requires more tuning and optimization efforts to improve data and memory access than for managing parallelism. The situation is further complicated by the recent expansion of the memory hierarchy, which is becoming deeper and more diversified with the adoption of new memory technologies and architectures such as 3D-stacked memory, non-volatile random-access memory (NVRAM), and hybrid software and hardware caches.

The authors believe it is important to elevate the notion of memory-centric programming, with relevance to the compute-centric or data-centric programming paradigms, to utilize the unprecedented and ever-elevating modern memory systems. *Memory-centric programming refers to the notion and techniques of exposing hardware memory system and its hierarchy, which could include DRAM and NUMA regions, shared and private caches, scratch pad, 3-D stacked memory, non-volatile memory, and remote memory, to the programmer via portable programming abstractions and APIs.* These interfaces seek to improve the dialogue between programmers and system software, and to enable compiler optimizations, runtime adaptation, and hardware reconfiguration with regard to data movement, beyond what can be achieved using existing parallel programming APIs. In this paper, we provide an overview of memory-centric programming concepts and principles for high performance computing.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; *Concurrent computing methodologies*; • **Theory of computation** → *Concurrency*;

KEYWORDS

Memory-Centric Programming, Abstract Machine Model, Explicit Data Mapping, Data Consistency

ACM Reference Format:
Yonghong Yan, Ron Brightwell, and Xian-He Sun. 2017. Principles of Memory-Centric Programming for High Performance Computing. In *MCHPC'17: Workshop on Memory-Centric Programming for HPC, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3145617.3158212

1 INTRODUCTION

The memory wall challenge – the growing disparity between CPU speed and memory speed – has been one of the most critical and long-standing challenges in computing. To combat the memory wall, both hardware and software techniques have been extensively developed for hiding latency and improving bandwidth utilization, including multi-level cache architectures, memory interleaving, software and hardware prefetching, software locality optimizations, and vector and hardware thread parallelism. While these techniques have been proven to be effective in containing the memory wall, the results vary significantly between applications and architectures [7, 9–12, 21]. For high performance computing, programming to achieve efficient execution of parallel applications often requires more tuning and optimization efforts to improve data and memory access than for managing parallelism, which often results in creating applications that are not performance-portable.

More recently, memory systems have become another wave of increasing complexity in computing, on top of the dramatically increased parallelism and heterogeneity in hardware. New memory technologies and architectures have been introduced into the conventional memory hierarchy, e.g. 3D-stacked memory [17, 22], NVRAM [18], and hybrid software/hardware cache architectures [16, 19]. In Table 1, we list in the last three rows the three main memory technologies that are recently being made into commercial products. These technologies and architecture advances improve memory performance (latency, bandwidth, and power consumption) and provide more options for users to optimize memory-intensive applications for optimal performance; however, they require significant programming efforts to use them efficiently.

For HPC, the performance challenge of parallel applications is often concerned with the tension between programmability, efficiency, and complexity when using these memory systems. A portable parallel programming framework that addresses the existing memory wall challenge and is able to sustain the emergence of new memory technologies and architectures for parallel processing is still needed.

*The authors were organizers of the MCHPC'17 workshop, which was held in conjunction with the International Conference on High Performance Computing, Networking, Storage and Analysis, 2017. The website for the workshop is http://passlab.github.io/mchpc/mchpc2017.html.

MCHPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5131-7/17/11…$15.00
https://doi.org/10.1145/3145617.3158212

Memory Types	Read Latency (ns)	Write Latency (ns)	Bandwidth (GB/s)	Dynamic Power	Leak Power	Density	Addressability
SRAM and Cache (L1, L2 and L3)	2 - 8	2 - 8		Low	High	10s MB	byte
DRAM	50 - 200	50 - 200	25	Medium	Medium	10s GB	Block/word
Stacked DRAM (HMC and HBM)	40 - 90	40 - 90	400	Low	Medium	10s GB	Block or Page
NAND NVRAM	100us	2-3ms	1GB/s for read and 10MB/s for write	Low for read; high for write	Low	100s GB	Block or Page
3D XPoint (Intel and Micron)	2-3x slower than DRAM	4-6x slower than DRAM				8-10x of DRAM	

Table 1: Comparison of memory technologies (data collected from Wikipedia and publications [1, 16, 25]): Stacked DRAM provides much higher bandwidth than DRAM. NVRAM and 3D XPoint have much higher write latency than read latency.

2 MEMORY-CENTRIC PROGRAMMING

Memory-centric programming, in contrast to compute-centric or data-centric programming paradigms, refers to the notion and techniques of exposing hardware memory system and its hierarchy, which could include DRAM and NUMA regions, shared and private caches, scratch pad, 3-D stacked memory, non-volatile memory and remote memory to the programmer via portable programming abstractions and APIs. These interfaces, such as for explicit memory allocation, data movement, and consistency enforcement between memories, enable explicit and fine-grained manipulation of data objects in different memories for extreme performance programming. These interfaces seek to improve the dialogue between programmers and system software, and to enable compiler optimizations, runtime adaptation, and hardware reconfiguration with regard to data movement, beyond what can be achieved using existing parallel programming APIs. The interfaces should also be abstract enough to sustain the emergence of new memory technologies and architectures for parallel processing, and also allow for architectural extensions for compiler to apply agreesive locality optimization.

This concept has been partially adopted in mainstream programming interfaces: place in OpenMP and X10, and locale in Chapel to represent memory regions in a system; shared modifier in CUDA and cache modifier in OpenACC for representing GPU scratch pad SRAM; the memkind library and the recent effort for OpenMP memory management for supporting 3-D stacked memory (HBM or HMC); and the PMEM library for persistent memory programming.

2.1 Comparison with Other Programming Models

The principles behind memory-centric parallel programming that differentiate it from compute- and data-centric programming are the use of portable interfaces for representing the memory hierarchy and for directly programming a memory system, and the notion and interface for explicit data placement, data movement, and consistency enforcement between memories and memory regions. In Figure 1, we compare the available parallel APIs and highlight the position of memory-centric programming in the taxonomy.

Memory-centric programming uses explicit shared data accesses and partitioning, as opposing to implicit data access of intra-node

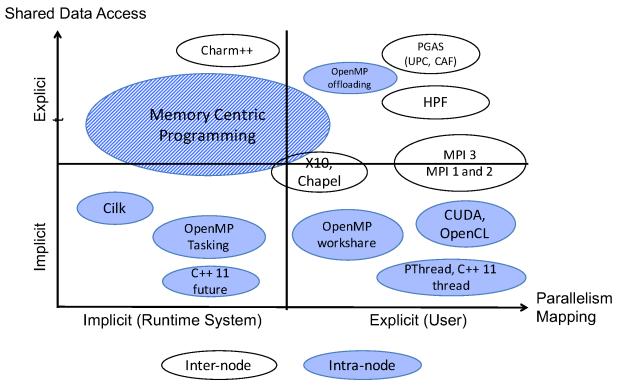


Figure 1: Classifying programming models with two taxonomies: shared data access (implicit or explicit) and managing/mapping parallelism (implicit or explicit)[1]

threading model such as OpenMP[2]. Implicit data sharing simplifies shared data access, promising better productivity than using explicit data access, yet more performance tuning and optimization efforts are needed for scaling beyond hundreds of cores [14, 20]. For comparison with inter-node or hybrid programming models, such as X10 [3], Chapel [2] and PGAS languages (HPF and UPC), the interfaces should be designed to differentiate vertical (caching) and horizontal (data copy) data movement in the memory hierarchy, and permit relaxed data consistency for achieving more aggressive latency hiding and data-computation overlapping than achievable by using the sequential data consistency model, which is assumed by most parallel programming models.

It is important to discuss the differences between memory-centric and data-centric programming such as Legion [8, 24]. In a Legion program, data and access to the data is organized using logical regions, which are mapped to hardware memory locations at runtime. Memory-centric programming provides APIs for directly mapping data to a memory system represented, thus eliminating the logical layer and reducing the overhead of the indirection.

3 MEMORY-CENTRIC PROGRAMMING PRINCIPLES

We have identified four principles for creating memory-centric parallel programming models.

[1] A similar comparison with four taxonomies (task identification, task mapping, data distribution, and communication mapping) was presented in the Berkeley View of Parallel Computing Landscape [6].

[2] OpenMP has shared and private clauses for annotating whether a new copy of a variable should be created or not for a thread. For shared-attribute data, the semantics of OpenMP still follow implicit data access approach.

3.1 Formal Abstract Machine Models

The foremost innovation of memory-centric programming is the use of a formal specification for representing computer system architecture including the memory systems across the programming software stack. An abstract machine model is one such option that provides conceptual models for hardware design, performance modeling, and compiler implementation [4, 5, 13, 15]; however, the model needs to be in a formal specification for parallel programming. For the hardware memory hierarchy, a tree-based abstract machine model is a natural choice. For example, our preliminary work of the hierarchical place tree (HPT) [27] has been used as a portable interface for programming tasks and data movement. An HPT for a real machine represents a necessary subset of the machine attributes that are important for parallel programming and performance optimization according to specific requirements. To illustrate this for a 4 CPU NUMA node (e.g. the 24-core Cray XE6), we created three distinct HPTs shown in Figure 2:a-c. Figure 2:c gives a flat HPT abstraction in which each core has uniform and direct access to the main memory. Figure 2:a, in contrast, gives a full abstraction of the memory system, including the interconnect. Based on this view, optimizations with respect to both the node architecture (e.g. improving locality through shared L3 cache) and the core microarchitecture (e.g. loop tiling according to the L2/L1 cache size) may be performed. Figure 2:b presents an intermediate view where caches and the interconnect are hidden, but not the NUMA region. Figure 2:d shows how an existing heterogeneous platform could be abstracted using the HPT model. In Figure 2:e, we show a sample design of the HPT APIs for memory-centric programming to implement hybrid tiling and parallelism for matrix multiplication. The distribute construct describes horizontal data distribution of the mapped arrays between sibling places.

An HPT can be parameterized differently for programmers, the compiler, and the runtime system to expose only the aspects of the machine that are important or relevant to each. A formal specification and usage model will significantly help produce portable code and improve program readability for architecture-specific optimizations. For example, memory optimization techniques for improving cache data locality, e.g. tiling, can be formally specified in a program using the HPT and explicit data mapping interfaces. For the compiler and the runtime, a parameterized HPT can be used for machine-aware compilation and data/locality-aware scheduling.

3.2 Explicit Data Operations and Memory Association

Using imperative programming languages, data has its program or domain view (e.g. array or scalar variable declaration). It then is associated with virtual memory, e.g. through heap memory allocation. Physical memory association happens often upon data accesses, a typical case of first-touch policy. Data access incur both physical page allocation and the actual read and write, which are all part of computation statements and implicit. The design principles for data placement, mapping and access in memory-centric programming are to make those operations explicit by using either new interfaces or annotating existing operations. Source and destination memories or regions, which are represented using the abstract machine model, should be specified in those interfaces to allow for direct manipulation of data objects in memory. These interfaces could include APIs and annotations for 1) associating domain view of data with memory storages when declaring data and moving data, 2) making data handling operations explicit by annotating it with designed interfaces, and 3) binding computation with data that is also known its memory location. As an example of this principle, Figure 3 shows our work of extending OpenMP for specifying the distribution of data and loops, and the alignment between data and loop iterations for CPU, GPU and Intel Xeon Phi accelerators [26]. The results (as many as 3X speedup) show that when being given those details about data and computation mapping, the runtime is able to chunk parallel loops and partition array data so that overlapped processing of data copy and computation can be effectively achieved, thus hiding the latency of data movement between CPU and accelerator memories. Thus, we believe such an approach or similar will be effective for programming the emerging memory architectures including the hybrid HBM cache system in Xeon Phi, and the page-based NVRAM software cache memory.

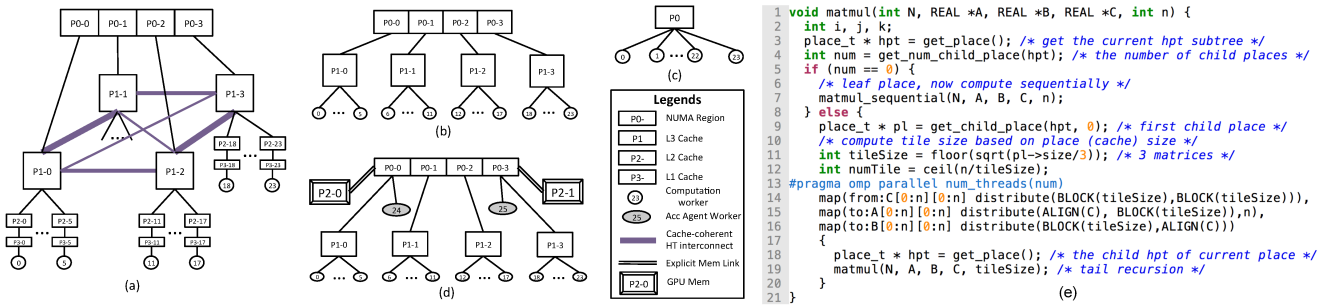
3.3 Relaxed Functional Semantics of Parallel Work Units

Work units are parallel scheduling tasks that are created from the specification of data parallelism, task parallelism, and offloading tasks. Work units in threading programming model [23] assume implicit data access and shared memory semantics, which limits the compiler and runtime for assisting runtime execution for overlapping computation and data movement. Relaxed functional semantics restrict that the interaction of a work unit with its external environment has to be explicitly specified, even through shared memory access. For example, a task should explicitly specifying its intended data operations on an external array (read and/or write). This is important for the automatic generation of task dependency graph, and the annotated read/write information will facilitate compiler optimization and improve runtime adaptation.

The introduction of `map, copyin, copyout` clause in OpenMP and OpenACC for accelerators are the approach for offloading computation since accelerators are in different memory space from host. By extending the similar semantics to programming on hybrid shared and discrete memory space will create portable program for both homogeneous and heterogeneous memory systems as well as providing more information to compiler and runtime systems for data aware scheduling and optimizations.

3.4 Relaxed Data Consistency and Coherence Enforcement in Different Granularity

To fully acknowledge the speed gaps between different memory technologies of modern memory systems, parallel programming models that permit relaxed data consistency will allow for applying more aggressive latency hiding and data-computation overlapping techniques than that by the sequential data consistency model, which is assumed by most parallel programming models. Also in the explicit data mapping principles introduced before, it enforces strict memory consistency, i.e., data movement happens at the location where the mapping is specified. Such a consistency model limits the amount of overlapping that can be achieved for hiding latency. To enable bulk data prefetching and out-of-order data movement and




```c
void matmul(int N, REAL *A, REAL *B, REAL *C, int n) {
  int i, j, k;
  place_t * hpt = get_place(); /* get the current hpt subtree */
  int num = get_num_child_place(hpt); /* the number of child places */
  if (num == 0) {
    /* leaf place, now compute sequentially */
    matmul_sequential(N, A, B, C, n);
  } else {
    place_t * pl = get_child_place(hpt, 0); /* first child place */
    /* compute tile size based on place (cache) size */
    int tileSize = floor(sqrt(pl->size/3)); /* 3 matrices */
    int numTile = ceil(n/tileSize);
#pragma omp parallel num_threads(num)
    map(from:C[0:n][0:n] distribute(BLOCK(tileSize),BLOCK(tileSize))),
    map(to:A[0:n][0:n] distribute(ALIGN(C), BLOCK(tileSize)),n),
    map(to:B[0:n][0:n] distribute(BLOCK(tileSize),ALIGN(C)))
    {
      place_t * hpt = get_place(); /* the child hpt of current place */
      matmul(N, A, B, C, tileSize); /* tail recursion */
    }
}
```
(e)

Figure 2: a, b, c: The HPTs for a 4-CPU NUMA machine, d: The HPT for a machine with 2 accelerators, and e: hybrid tiling with parallelism for matrix multiplication using the initial HPT APIs

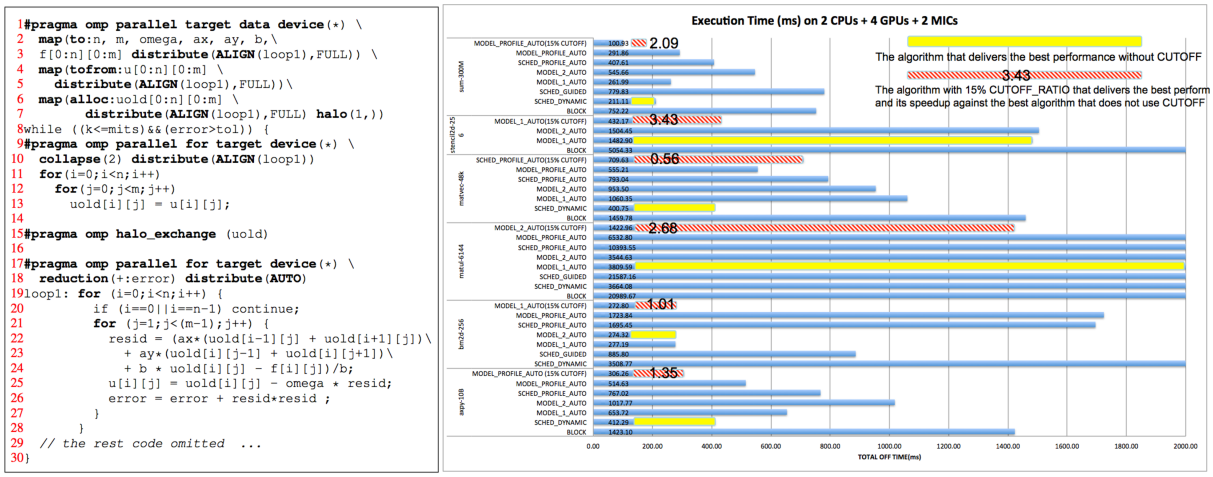


Figure 3: The use of distribute **and** AUTO **and** ALIGN **clauses to explicitly specify data mapping, distribution and binding with loop distributions [26]. Left figure: Jacobi example; Right figure: performance results using different loop distribution policies.**

computation, the mapping interfaces needs to have more relaxed semantics.

Thus memory-centric programming should enable relaxed data consistency and coherence enforcement for users to explore high degree of computation and data movement overlapping. 1) The model does not enforce ordering of read/write of multiple regions mapped from the same memory segment, or of even the same physical memory segment, thus allowing overlapping read/write and write/write to the same memory segment. 2) Memory consistency enforcements are mostly implemented as hardware memory barriers, which strictly forces the ordering of all the load and store operations across the barrier point. The relaxed data consistency model, if implemented using software, will allow for read-write ordering of specific locations, thus eliminating the overcommittment of ordering that may limit the amount of overlapping of data movement. 3) If implemented as software managed, the model should also allow for consistency and coherence in different data size, thus the granularity. 4) Data consistency can be applied between discrete memory spaces, such as between CPU and GPU memory, while memory consistency is for shared memory.

4 CONCLUSION

In this position paper, we propose memory-centric parallel programming concepts for high performance computing, focusing on addressing the challenges of existing and emerging memor systems for extreme performance programming. We identified four principles for realizing memory-centric programming interfaces and techniques. Our intention is to elevate the notion of memory-centric programming in mainstream programming model and compiler research for parallel computing. We believe it lays the foundation for addressing the memory challenges in system software.

There are still many open problems. A major one is in the selection of language features considering the trade-off between programing complexity, difficulty of compiler and runtime implementations, and performance. The number of details exposed to programmers is the trade-off between programmability and performance. The heuristics depend heavily on the quality of the compiler and runtime system implementation, as well as the application. Should the interface design follow revolutionary approach by creating new programming languages, or evolutionary principles by extending existingh parallel programming standard? It is a critical question when considering migrating existing parallel applications. As for

the data consistency model, should we choose different models according the type of memory for which we are enforcing consistency? How should software-implemented consistency and coherence models work with hardware support for memory consistency?

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1652732 and 1536079, and by Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] [n. d.]. ISSCC 2016 TechTrends. ([n. d.]). http://isscc.org/doc/2016/ISSCC2016_TechTrends.pdf.

[2] [n. d.]. The Chapel Parallel Programming Language. http://chapel.cray.com/. ([n. d.]).

[3] [n. d.]. X10: Performance and Productivity at Scale. http://x10-lang.org/. ([n. d.]).

[4] B. Alpern, L. Carter, and J. Ferrante. 1993. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings.* 116–123. https://doi.org/10.1109/PMMP.1993.315548

[5] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. 2014. Abstract Machine Models and Proxy Architectures for Exascale Computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing (Co-HPC '14)*. IEEE Press, Piscataway, NJ, USA, 25–32. https://doi.org/10.1109/Co-HPC.2014.4

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67. https://doi.org/10.1145/1562764.1562783

[7] Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. 2004. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism* 6, 7 (2004).

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[9] Douglas C. Burger, James R. Goodman, and Alain KÃđgi. 1995. *The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors.* Technical Report. University of Wisconsin-Madison Computer Sciences.

[10] Surendra Byna, Yong Chen, and Xian-He Sun. 2008. A Taxonomy of Data Prefetching Mechanisms. In *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN '08)*. IEEE Computer Society, Washington, DC, USA, 19–24. https://doi.org/10.1109/I-SPAN.2008.24

[11] Laura Carrington, Allan Snavely, and Nicole Wolter. 2006. A Performance Prediction Framework for Scientific Applications. *Future Gener. Comput. Syst.* 22, 3 (Feb. 2006), 336–346. https://doi.org/10.1016/j.future.2004.11.019

[12] Francky Catthoor, Nikil D. Dutt, and Christoforos E. Kozyrakis. 2000. How to Solve the Current Memory Access and Data Transfer Bottlenecks: At the Processor Architecture or at the Compiler Level. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '00)*. ACM, New York, NY, USA, 426–435. https://doi.org/10.1145/343647.343813

[13] Stephan Diehl and Peter Sestoft. 2000. Abstract Machines for Programming Language Implementation. *Future Gener. Comput. Syst.* 16, 7 (May 2000), 739–751. https://doi.org/10.1016/S0167-739X(99)00088-6

[14] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. 2005. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, Washington, DC, USA, 35–. https://doi.org/10.1109/SC.2005.53

[15] M. Kara, J. R. Davy, D. Goodeve, and J. Nash (Eds.). 1997. *Abstract Machine Models for Parallel and Distributed Computing.* IOS Press, Amsterdam, The Netherlands, The Netherlands.

[16] Suji Lee, Jongpil Jung, and Chong-Min Kyung. 2012. Hybrid cache architecture replacing SRAM cache with future memory technology. In *2012 IEEE International Symposium on Circuits and Systems*. IEEE, 2481–2484.

[17] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 453–464. https://doi.org/10.1109/ISCA.2008.15

[18] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. 2014. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters* 9, 1 (2014), 1–33. https://doi.org/10.1186/1556-276X-9-526

[19] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2015), 1524–1537.

[20] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. 2013. Benchmarking Usability and Performance of Multicore Languages. *CoRR* abs/1302.2837 (2013). http://arxiv.org/abs/1302.2837

[21] S. S. Nemawarkar and G. R. Gao. 1997. Latency tolerance: a metric for performance analysis of multithreaded architectures. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*. 227–232. https://doi.org/10.1109/IPPS.1997.580899

[22] P. Ramm, A. Klumpp, J. Weber, N. Lietaer, M. Taklo, W. De Raedt, T. Fritzsch, and P. Couderc. 2010. 3D Integration technology: Status and application development. In *ESSCIRC, 2010 Proceedings of the*. 9–16. https://doi.org/10.1109/ESSCIRC.2010.5619857

[23] S. Salehian, Jiawen Liu, and Yonghong Yan. 2017. Comparison of Threading Programming Models. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 766–774. https://doi.org/10.1109/IPDPSW.2017.141

[24] Sean Treichler, Michael Bauer, and Alex Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 495–514. https://doi.org/10.1145/2509136.2509545

[25] Yuan Xie. 2011. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers* 1 (2011), 44–51.

[26] Yonghong Yan, Jiawen Liu, Kirk W. Cameron, and Mariam Umar. 2017. HOMP: Automated Distribution of Parallel Loops and Data in Highly Parallel Accelerator-Based Systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 788–798. https://doi.org/10.1109/IPDPS.2017.99

[27] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement.. In *LCPC'09*. 172–187.