On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking [*]

David A. Bader, Virat Agarwal, Kamesh Madduri

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA
{bader, virat, kamesh}@cc.gatech.edu

Abstract

The Sony-Toshiba-IBM Cell Broadband Engine is a heterogeneous multicore architecture that consists of a traditional microprocessor (PPE), with eight SIMD co-processing units (SPEs) integrated on-chip. We present a complexity model for designing algorithms on the Cell processor, along with a systematic procedure for algorithm analysis. To estimate the execution time of the algorithm, we consider the computational complexity, memory access patterns (DMA transfer sizes and latency), and the complexity of branching instructions. This model, coupled with the analysis procedure, simplifies algorithm design on the Cell and enables quick identification of potential implementation bottlenecks. Using the model, we design an efficient implementation of list ranking, a representative problem from the class of combinatorial and graph-theoretic applications. Due to its highly irregular memory patterns, list ranking is a particularly challenging problem to parallelize on current cache-based and distributed memory architectures. We describe a generic work-partitioning technique on the Cell to hide memory access latency, and apply this to efficiently implement list ranking. We run our algorithm on a 3.2 GHz Cell processor using an IBM QS20 Cell Blade and demonstrate a substantial speedup for list ranking on the Cell in comparison to traditional cache-based microprocessors. For a random linked list of 1 million nodes, we achieve an an overall speedup of 8.34 over a PPE-only implementation.

[*]This work was supported in part by NSF Grants CNS-0614915, CAREER CCF-0611589, ITR EF/BIO 03-31654, and DARPA Contract NBCH30390004.

1. Introduction

The Cell Broadband Engine (or the Cell BE) [16] is a novel architectural design by Sony, Toshiba, and IBM (STI), primarily targeting high performance multimedia and gaming applications. It is a heterogeneous multicore chip that is significantly different from conventional multi-processor or multicore architectures. It consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The EIB supports a peak bandwidth of 204.8 GBytes/s for intrachip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GBytes/s to main memory. The I/O controller provides peak bandwidths of 25 GBytes/s inbound and 35 GBytes/s outbound. Kistler et al. [17] analyze the communication network of the Cell processor and state that applications that rely heavily on random scatter and or gather accesses to main memory can take advantage of the high communication bandwidth and low latency. We refer the reader to [18, 10, 17, 13, 7] for additional details. The Cell is used in Sony's PlayStation 3 gaming console, Mercury Computer System's dual Cell-based blade servers, and IBM's QS20 Cell Blades.

There are several unique architectural features in Cell that clearly distinguish it from current microprocessors. The Cell chip is a computational workhorse; it offers a theoretical peak single-precision floating point performance of 204.8 GFlops/sec (assuming the current clock speed of 3.2 GHz). We can exploit parallelism at multiple levels on the Cell, each chip has eight SPEs, with two-way instruction-level parallelism on each SPE. Further, the SPE supports both scalar as well as single-instruction, multiple data (SIMD) computations [14]. Also, the on-chip coherent bus and interconnection network elements have been specially designed to cater for high performance on bandwidth-intensive applications (such as those in gaming and multimedia). The custom, modular system-on-a-chip implementation results in a power-efficient design.

All these features make the Cell attractive for scientific computing, as well as an alternative architecture for general purpose computing. Williams et al. [22] recently analyzed the performance of Cell for key scientific kernels such as dense matrix multiply, sparse matrix vector multiply and 1D and 2D fast Fourier transforms. They demonstrate that the Cell performs impressively for applications with predictable memory access patterns, and that on-chip communication and SPU computation can be overlapped more effectively on the Cell than on conventional cache-based approaches.

The heterogeneous processors, limited on-chip memory and multiple avenues of parallelism on the Cell processor make algorithm design and implementation a new challenge. Analyzing algorithms using traditional sequential complexity models like the RAM model fail to account for all of the Cell architectural intricacies. There is currently no simple and accepted model of computation for the Cell, and in general, for multicore architectures. Our primary contribution in this paper is a *complexity model* for the design and analysis of parallel algorithms on the Cell architecture. Taking most unique architectural features of the Cell into account, we come up with a simple model that estimates the execution time. We express the algorithm complexity on the Cell processor using the triplet $\langle T_C, T_D, T_B \rangle$, where T_C denotes the computational complexity, T_D the number of DMA requests, and T_B the number of branching instructions, all expressed in terms of the problem size. We explain the rationale behind the choice of these three parameters in Section 2. We then present a *systematic methodology* for analyzing algorithms using this complexity model, and illustrate with a simple example such as matrix multiplication.

It is general perception that the Cell architecture is not suited for problems that involve fine-grained memory accesses, and where there is insufficient computation to hide memory latency. The *list ranking problem* [8, 20, 15] is representative of such problems, and is a fundamental paradigm for the design of many parallel combinatorial and graph-theoretic applications. Using list ranking, we have designed fast parallel algorithms for shared memory computers and demonstrated speedups compared with the best sequential implementation for graph theoretic problems such as ear decomposition [4], tree contraction and expression evaluation [5], spanning tree [1] and minimum spanning forest [2]. Given an arbitrary linked list that is stored in a contiguous area of memory, the list ranking problem determines the distance from each node to the head of the list. Due to lack of locality in this problem, it is difficult to obtain parallel speedup, and no efficient distributed memory implementation exists. We have an implementation of this problem that obtains significant speedup using our new techniques for latency tolerance and load balancing on Cell. We present these technique in detail in Section 4.2.

We also present an experimental study of list ranking, comparing our Cell code with implementations on current microprocessors and high-end shared memory and multithreaded architectures. Our main results are summarized here:

- Our latency-hiding technique, boosts our Cell performance by a factor of about 4.1 for both random and ordered lists.
- Through the tuning of one parameter in our algorithm, our list ranking code is load-balanced across the SPEs with high probability, even for random lists.
- The Cell achieves an average speedup of 8 over the performance on current cache-based microprocessors (for input instances that do not fit into the L2 cache).
- On a random list of 1 million nodes, we obtain a speedup of 8.34 compared to a single-threaded PPE-only implementation. For an ordered list (with stride-1 accesses only), the speedup over a PPE-only implementation is 1.56.

2. A complexity model for the Cell architecture

The Cell Broadband Engine (CBE) [13] consists of a traditional microprocessor (PPE) that controls eight Synergistic processing units (SPEs), a high speed memory controller, and a high bandwidth bus interface (EIB). There are several architectural features that can be exploited for performance.

- The (SPEs) are designed as compute-intensive co-processors, while the PowerPC unit (the PPE) orchestrates the control flow. So it is necessary to partition the computation among the SPEs, and an efficient SPE implementation should also exploit the SIMD instruction set.
- The SPEs operate on a limited on-chip memory (256 KB local store) that stores both instructions and data required by the program. Unlike the PPE, the SPE cannot access memory directly, but has to transfer data

and instructions using asynchronous coherent DMA commands. Algorithm design must account for DMA transfers (i.e., the latency of DMA transfers, as well as their frequency) which may be a significant cost.

- The SPE also differs from conventional microprocessors in the way branches are handled. The SPE does not include dynamic branch prediction, but instead relies on compiler-generated branch hints to improve instruction prefetching. Thus, there is a significant penalty associated with branch misprediction, and branching instructions should be minimized for designing an efficient implementation.

We present a complexity model to simplify the design of parallel algorithms on the Cell. Let n denote the problem size. We model the execution time using the triplet $\langle T_C, T_D, T_B \rangle$, where T_C denotes the computational complexity, T_D the number of DMA requests, and T_B the number of branching instructions. We consider the computation on the SPEs ($T_{C,SPE}$) and PPE ($T_{C,PPE}$) separately, and T_C denotes the sum of these terms. $T_{C,SPE}$ is the maximum of $T_{C,SPE(i)}$ for $1 \leq i \leq p$, where p is number of SPEs. In addition, we have T_D, an upper bound on the number of DMA requests made by a single SPE. This is an important parameter, as the latency due to a large number of DMA requests might dominate over the actual computation. In cases when the complexity of $T_{C,SPE}$ dominates over T_D, we can ignore the overhead due to DMA requests. Similarly, branch mispredictions constitute a significant overhead. Since it may be difficult to compute the actual percentage of mispredictions, we just report the asymptotic number of branches in our algorithm. For algorithms in which the misprediction probability is low, we can ignore the effects of branching.

Our model is similar to the Helman-JáJá model for SMPs [11] in that we try to estimate memory latency in addition to computational complexity. Also, our model is more tailored to heterogeneous multicore systems than general purpose parallel computing models such as LogP [9], BSP [21] and QSM [19]. The execution time is dominated by the SPE that does the maximum amount of work. We note that exploiting the SIMD features results in only a constant factor improvement in the performance, and does not affect the asymptotic analysis. This model does not take into account synchronization mechanisms such as on-chip mailboxes, and SPU operation under the isolated mode. Also, our model does not consider the effect of floating-point precision on the performance of numerical algorithms, which can be quite significant [22].

3. A Procedure for Algorithm Analysis

We now discuss a systematic procedure for analyzing algorithms using the above model:

1. We compute the computational complexity $T_{C,SPE}$.
2. Next, we determine the complexity of DMA requests T_D in terms of the input parameters.
   - If the DMA request complexity is a constant, then typically computation would dominate over memory accesses and we can ignore the latency due to DMA transfers.
   - Otherwise, we need to further analyze the algorithm, taking into consideration the size of DMA transfers, as well as the computational granularity.
3. It is possible to issue non-blocking DMA requests on the SPE, and so we can keep the SPE busy with computation while waiting for a DMA request to be completed. However, if there is insufficient computation in the algorithm between a DMA request and its completion, the SPE will be idle. We analyze this effect by computing the *computational complexity in the average case* between a DMA request and its completion. If this term is a function of the input parameters, this implies that memory latency can be hidden by computation.
4. Finally, we compute the number of branching instructions T_B in the algorithm. These should be minimized as much as possible in order to design an efficient algorithm.

We present a simple example to illustrate the use of our model, as well as the above algorithm analysis procedure.

Matrix multiplication ($C = A * B$, $c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$, for $1 \leq i, j \leq n$) on the Cell is analyzed as follows:

- We partition computation among the eight SPEs by assigning each SPE $\frac{n}{p}$ rows of A and entire B.
- Let us assume that each SPE can obtain b rows of A and b columns of B in a single DMA transfer. Thus, we would require $O(\frac{n^2}{b^2})$ DMA transfers, and T_D is $O(\frac{n^2}{pb^2})$.

Chen et al. describe their Cell implementation of this algorithm in [7]. The algorithmic complexity is given by $T_C = O(\frac{n^3}{p})$, $T_D = O(\frac{n^2}{pb^2})$ and $T_B = O(n^2)$. Using the analysis procedure, we note that the DMA request complexity is not a constant. Following step 3, we compute the average case of the computational complexity between the

DMA request and its completion, assuming non-blocking DMAs and double buffering. This is given by $O(nb^2)$ (as the complexity of computing b^2 elements in C is $O(n)$). Thus, we can ignore the constant DMA latency for each transfer, and the algorithm running time is dominated by computation for sufficiently large n. However, note that we have $O(n^2)$ branches due to the absence of a branch predictor on the SPE, which might degrade performance if they result in mispredicts. Using SIMD features, it is possible to achieve a peak $CPI\ of$ 0.5. Chen et al. in fact obtain a CPI of 0.508 for their implementation of the above algorithm, incorporating optimizations such as SIMD, double-buffering and software pipelining.

Algorithm design and analysis is more complex for irregular, memory-intensive applications, and problems exhibiting poor locality. *List ranking* is representative of this class of problems, and is a fundamental technique for the design of several combinatorial and graph-theoretic applications on parallel processors. After a brief introduction to list ranking in the next section, we describe our design of an efficient algorithm for the Cell.

4. List Ranking using Cell

Given an arbitrary linked list that is stored in a contiguous area of memory, the list ranking problem determines the distance of each node to the head of the list. For a random list, the memory access patterns are highly irregular, and this makes list ranking a challenging problem to solve efficiently on parallel architectures. Implementations that yield parallel speedup on shared memory systems exist [11, 3], yet none are known for distributed memory systems.

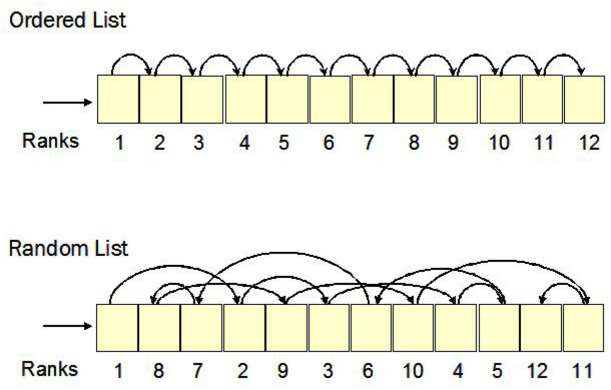

Figure 1. List ranking for ordered and random list

4.1. Parallel Algorithm

List ranking is an instance of the more general prefix problem [3]. Let X be an array of n elements stored in arbitrary order. For each element i, let $X(i).value$ denote its value and $X(i).next$ the index of its successor. Then for any binary associative operator $\oplus$, compute $X(i).prefix$ such that $X(head).prefix = X(head).value$ and $X(i).prefix = X(i).value \oplus X(predecessor).prefix$, where *head* is the first element of the list, i is not equal to *head*, and *predecessor* is the node preceding i in the list. If all values are 1 and the associative operation is addition, then prefix reduces to list ranking. We assume that we know the location of the head h of the list, otherwise we can easily locate it. The parallel algorithm for a canonical parallel computer with p processors is as follows:

1. Partition the input list into s sublists by randomly choosing one node from each memory block of $n/(s-1)$ nodes, where s is $\Omega(p \log n)$. Create the array *Sublists* of size s.

2. Traverse each sublist computing the prefix sum of each node within the sublists. Each node records its sublist index. The input value of a node in the *Sublists* array is the sublist prefix sum of the last node in the previous *Sublists*.

3. The prefix sums of the records in the *Sublists* array are then calculated.

4. Each node adds its current prefix sum value (value of a node within a sublist) and the prefix sum of its corresponding *Sublists* record to get its final prefix sums value. This prefix sum value is the required label of the leaves.

We map this to Cell and analyze it as follows. Assume that we start with eight sublists , one per SPE. Using DMA fetches, the SPEs keep obtaining the successor elements until they reach a sublist end, or the end of the list. Analyzing the complexity of this algorithm using our model, we have $T_C = O(\frac{n}{p})$, $T_D = O(\frac{n}{p})$ and $T_B = O(1)$. From step 2 of the procedure, since the complexity of DMA fetches is a function of n, we analyze the computational complexity in the average case between a DMA request and its completion. This is clearly $O(1)$, since we do not perform any significant computation while waiting for the DMA request to complete. This may lead to processor stalls, and since the number of DMA requests is $O(n)$, stall cycles might dominate the optimal $O(n)$ work required for list ranking. Our asymptotic analysis offers only a limited insight into the algorithm, and we have to inspect the algorithm at the instruction-level and design alternative approaches to hide DMA latency.

4.2. A Novel Latency-hiding Technique for Irregular applications

Due to the limited local store (256 KB) within a SPE, memory-intensive applications that have irregular memory access patterns require frequent DMA transfers to fetch the data. The relatively high latency of a DMA transfer creates a bottleneck in achieving performance for these applications. Several combinatorial problems, such as the ones that arise in graph theory, belong to this class of problems. Formulating a general strategy that helps overcome the latency overhead will provide direction to the design and optimization of irregular applications on Cell.

Since the Cell supports non-blocking memory transfers, memory transfer latency will not be a problem if we have sufficient computation between a request and completion. However, if we do not have enough computation in this period (for instance, the Helman-JáJá list ranking algorithm), the SPE will stall for the request to be completed. A generic solution to this problem would be to restructure the algorithm such that the SPE keeps doing useful computation until the memory request is completed. This essentially requires identification of an additional level of parallelism/concurrency within each SPE. Note that if the computation can be decomposed into several independent tasks, we can overcome latency by exploiting concurrency in the problem.

Our technique is analogous to the concept of tolerating latency in modern architectures using thread-level parallelism. The SPE does not have support for hardware multithreading, and so we manage the computation through *software-managed threads*. The SPE computation is distributed to a set of software-managed threads (SM-Threads) and at any instant, one thread is active. We keep switching software contexts so that we do computation between a DMA request and its completetion. We use a round-robin schedule for the threads.

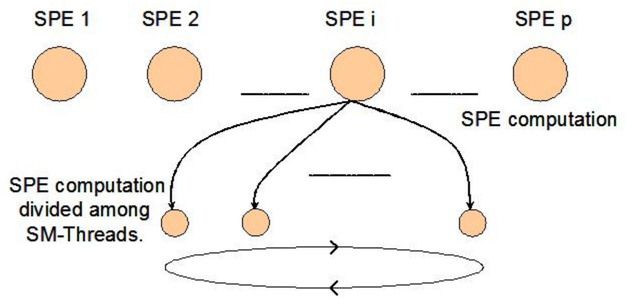

Figure 2. Illustration of the technique

Through instruction-level profiling, it is possible to determine the minimum number of SM-Threads that are needed to hide the memory latency. Note that utilizing more SM-Threads than required also incurs an overhead. Each SM-Thread introduces additional computation and also requires memory on the limited local store. Thus, we have a trade-off between the number of SM-Threads and latency due to DMA stalls. In the next section, we will use this technique to efficiently implement list ranking on Cell.

4.3. Implementation

Our Cell implementation (described in high-level in the following four steps) is similar to the Helman–JáJá algorithm. Let us assume p SPEs in the analysis.

1. We uniformly pick s *head nodes* in the list and assign them to the SPEs. So, each SPE will traverse s/p sublists.

2. Using these s/p sublists as independent SM-Threads, we adopt the latency-hiding technique. We divide the s/p sublists into b DMA list transfers. Using one DMA list transfer, we fetch the next elements for a set of s/pb lists. After issuing a DMA list transfer request, we move onto the next set of sublists and so forth, thus keeping the SPU busy until this DMA transfer is complete. Figure 3 illustrates step 3 of this algorithm.

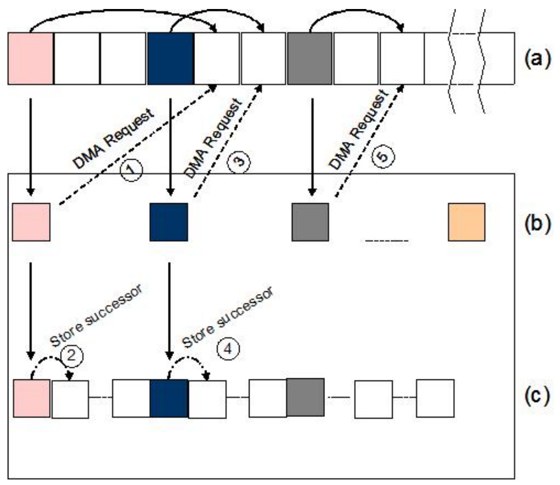

Figure 3. Step 2 of List ranking on Cell. (a) Linked list for which list ranking is to be done. Colored nodes here are allocated to SPE(i), (b) View from SPE(i), it has s/p sublist head nodes to traverse concurrently, (c) This array is used to store sublists in contiguous area of memory. When this gets full, we transfer it back to the main memory.

We maintain temporary structures in the Local Store (LS) for these sublists, so that the LS can create a contiguous sublist out of these randomly scattered sublists, by creating a chain of next elements for the sublists.

After one complete round, we manually revive this SM-Thread and wait for the DMA transfer to complete. Note that there will be no stall if we have sufficient number of SM-Threads (we determine this number in Sec. 4.4) to hide the latency. We store the elements that are fetched into the temporary structures, initiate a new DMA list transfer request for fetching the successors of these newly fetched elements, and move on to the next set of sublists.

When these temporary structures get full, we initiate a new DMA list transfer request to transfer back these elements to the main memory.

At the end of Step 2, we have the prefix sum of each node within the sublist for each sublist within the SPU. Also, we have the randomly scattered sublists stored into a contiguous area of memory.

3. Compute the rank of each sublist head node using the PPU.

The running time for step 2 of the algorithm dominates over the rest of algorithm by an order of magnitude. In the asymptotic notation, this step is O(n). It consists of an outer loop of O(s) and an inner loop of O(*length of the sublist*). Since the lengths of the sublists are different, the amount of work performed by each SM-Thread differs. For a large number of threads, we get sufficient computation for the SPE to hide DMA latency even when the load is imbalanced. Helman and JáJá [11, 12] established that with high probability, no processor would traverse more $\alpha(s)\frac{n}{p}$ elements for $\alpha(s) \geq 2.62$. Thus, the load is balanced among various SPEs under this constraint. In our implementation, we incorporate recommended software strategies [6] and techniques to exploit the architectural features of Cell. For instance, we use manual loop unrolling, double buffering, branch hints, and design our implementation for a limited local store.

4.4. Performance Results

We report our performance results from actual runs on a IBM BladeCenter QS20, with two 3.2 GHz Cell BE processors, 512 KB Level 2 cache per processor, and 1 GB memory (512 MB per processor). We use one processor for measuring performance and compile the code using the gcc compiler provided with Cell SDK 1.1, with level 3 optimization.

Similar to [11, 3] we use two classes of lists to test our code, *Ordered* and *Random*. An ordered list representation places each node in the list according to its rank. Thus node i is placed at position i, and its successor is at position $i+1$. A random list representation places successive elements randomly in the array.

Our significant contribution to this paper is a generic work partitioning technique to hide memory latency. We demonstrate the results of this technique for list ranking: we use DMA-level parallelism to vary the number of outstanding DMA requests on each SPE, as well as partition the problem and allocate more sublists to each SPE. Fig. 4 shows the performance boost we obtain as we tune the DMA parameter. From instruction level profiling of our code we determine that the exact number of computational clock cycles between a DMA transfer request and its completion are 75. Comparing this with the DMA transfer latency (90ns, i.e. about 270 clock cycles) suggests that four outstanding DMA requests should be sufficient for hiding the DMA latency. Our results confirm this analysis and we obtain an improvement factor of 4.1 using 8 DMA buffers.

In Fig. 5 we present the results for load balancing among the 8 SPEs, as the number of sublists are varied. For ordered lists, we allocate equal chunks to each SPE. Thus, load is balanced among the SPEs in this case. For random lists, since the length of each sublist varies, the work performed by each SPE varies. We achieve a better load balancing by increasing the number of sublists. Fig. 5 illustrates this: load balancing is better for 64 sublists than the case of 8 sublists.

We present a performance comparison of our implementation of list ranking on Cell with other single processor and parallel architectures. We consider both random and ordered lists with 8 million nodes.

Fig. 6 shows the running time of our Cell implementation compared with efficient implementations of list ranking on the following architectures:

Intel_x86: 3.2 GHz Intel Xeon processor, 1 MB L2 cache, Intel C compiler v9.1.

Intel_i686: 2.8 GHz Intel Xeon processor, 2 MB L2 cache, Intel C compiler v9.1.

Intel_ia64: 900 MHz Intel Itanium 2 processor, 256 KB L2 cache, Intel C compiler v9.1.

SunUS_III: 900 MHz UltraSparc-III processor, Sun C compiler v5.8.

MTA-[1,2,8]: 220 MHz Cray MTA-2 processor, no data cache. We report results for 1,2,8 processors.

SunUS-[1,2,8]: 400 MHz UltraSparc II Symmetric Multiprocessor system (Sun E4500), 4 MB L2 cache, Sun C compiler. We report results for 1,2 and 8 processors.

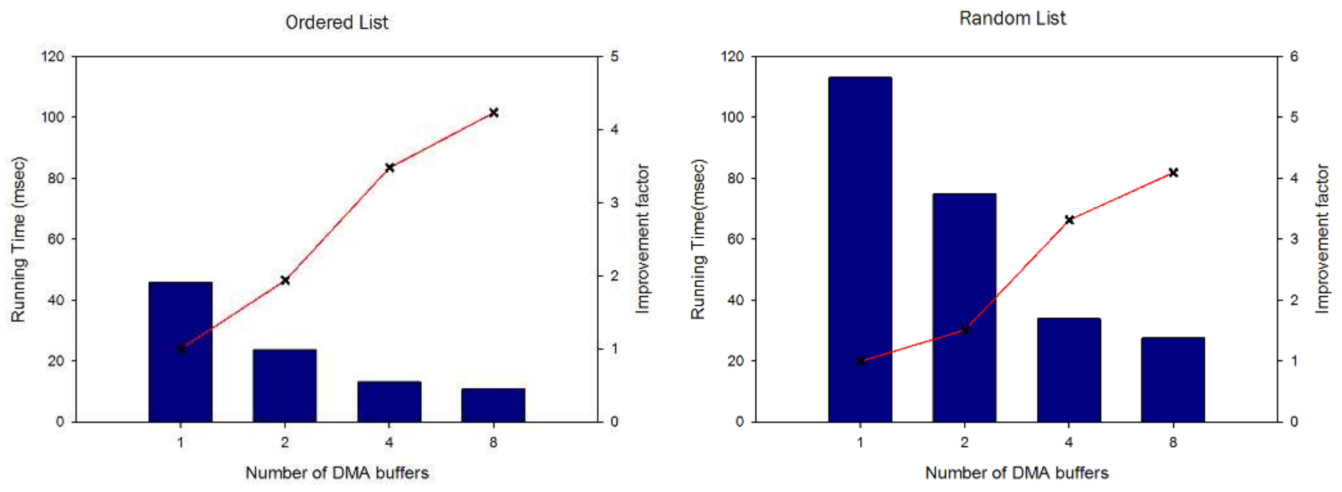

Figure 4. Achieving Latency Tolerance for Step 2 of the List ranking algorithm through DMA parameter tuning, for lists of size 2^{20}

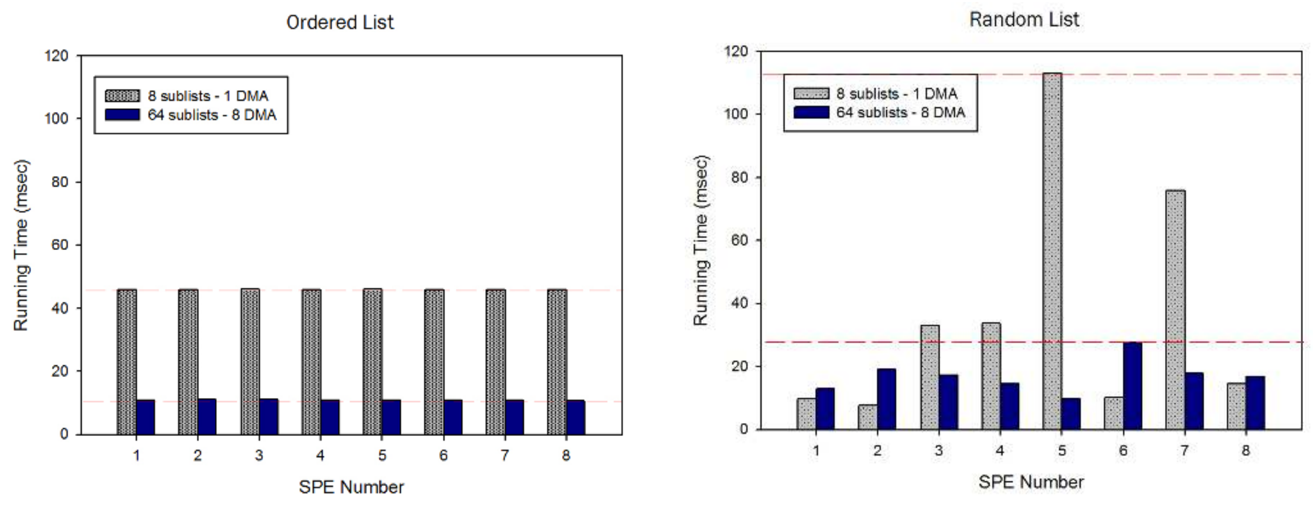

Figure 5. Load Balancing among SPEs for Step 2 of the List ranking algorithm for lists of size 2^{20}. The upper and lower dashed horizontal lines represent the running time of this step of the algorithm for 8 and 64 sublists respectively.

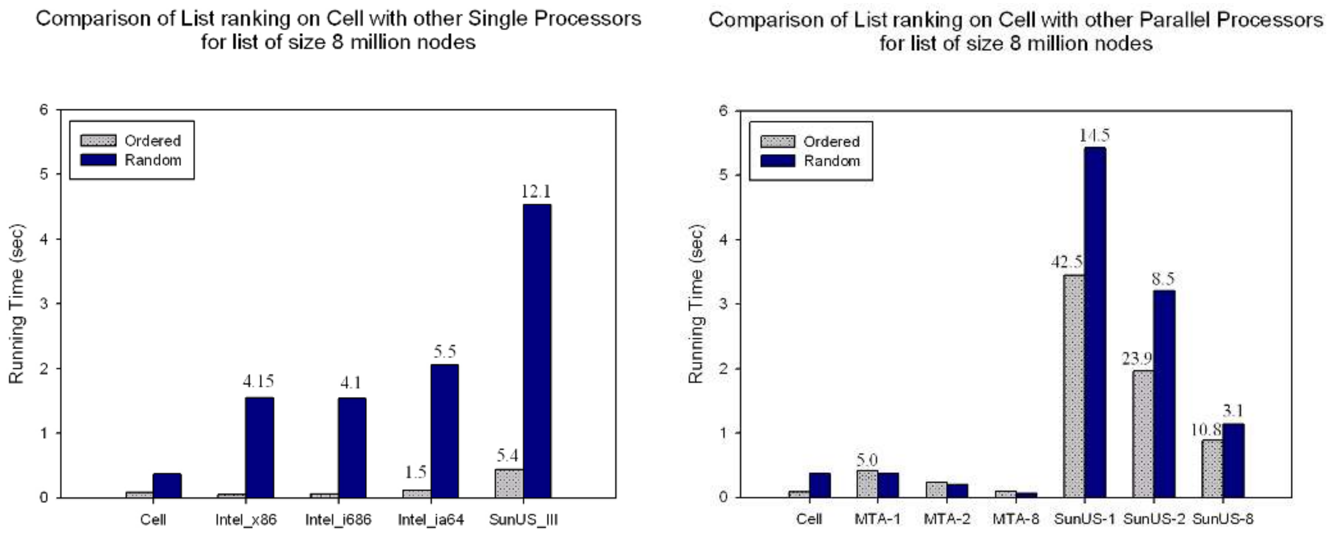

Figure 6. Performance of List ranking on Cell as compared to other single processor and parallel architectures for lists of size 8 million nodes. The speedup of the Cell implementation over the architectures is given above the respectives bars.

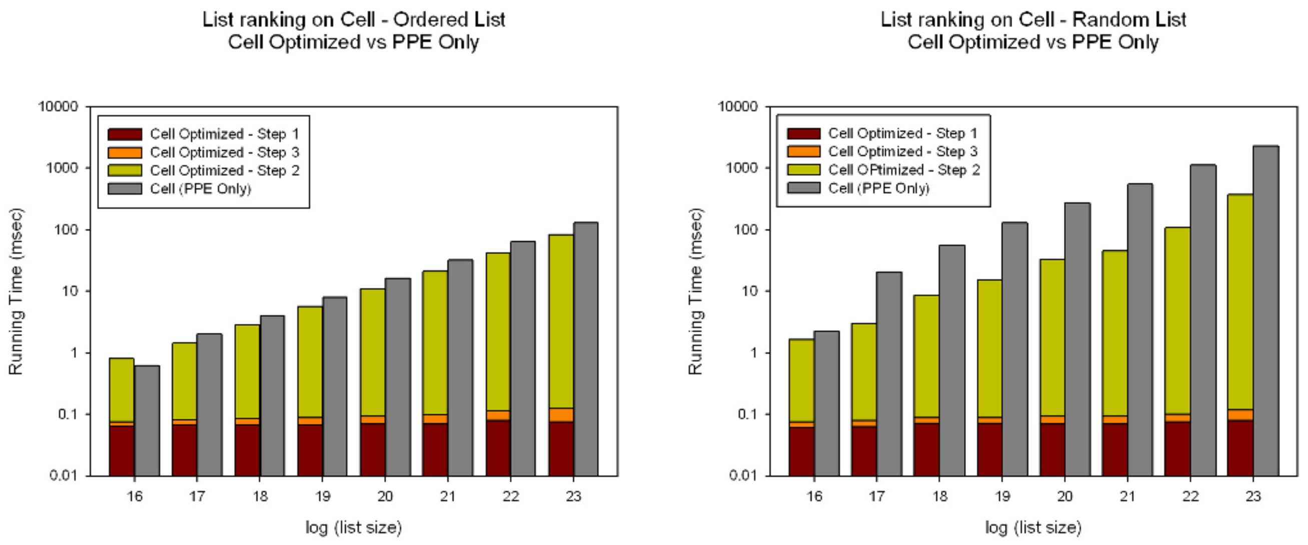

Figure 7. Performance Comparison of sequential implementation on PPE to our parallel implementation of List ranking on Cell for Ordered (Left) and Random (Right) lists

Finally, we demonstrate a substantial speedup of our Cell implementation over a sequential implementation using the PPE only. We compare a simple pointer-chasing approach to our algorithm using different problem instances. Fig. 7 shows that for random lists we get an overall speedup of 8.34 (1 million vertices), and even for ordered lists we get a speedup of 1.5.

5. Conclusions and Future Work

In summary, we present a complexity model to simplify the design of algorithms on the Cell Broadband Engine, and a systematic procedure to evaluate their performance. To estimate the execution time of an algorithm, we consider the computational complexity, memory access patterns (DMA transfer sizes and latency), and the complexity of branching instructions. This model helps identify potential bottlenecks in the algorithm. We also present a generic work partitioning technique to hide memory latency on Cell. This technique can be applied to many irregular algorithms having exhibiting unpredictable memory access patterns. Using this technique, we develop a fast parallel implementation of the list ranking algorithm for the Cell processor and confirm the efficacy of our technique by demonstrating an improvement factor of 4.1 as we tune the DMA parameter. Most importantly we demonstrate an overall speedup of 8.34 of our implementation over an efficient PPE-only sequential implementation. We show substantial speedups by comparing the performance of our list ranking implementation with several single processor and parallel architectures.

We thank Sidney Manning (IBM Corporation) and Vipin Sachdeva (IBM Research) for providing valuable inputs during the course of our research.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[2] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[3] D.A. Bader, G. Cong, and J. Feo. A comparison of the performance of list ranking and connected components algorithms on SMP and MTA shared-memory systems. Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, October 2004. Submitted for publication.

[4] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[5] D.A. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002. Springer-Verlag.

[6] D.A. Brokenshire. Maximizing the power of the Cell Broadband Engine Processor: 25 tips to optimal application performance. White paper, June 2006.

[7] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. IBM White paper, November 2005.

[8] R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344–352, 1989.

[9] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th Symp. Principles and Practice of Parallel Programming*, pages 1–12. ACM SIGPLAN, May 1993.

[10] B. Flachs and *et al.* A streaming processor unit for a Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 134–135, San Fransisco, CA, USA, February 2005.

[11] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[12] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[13] H.P. Hofstee. Cell Broadband Engine Architecture from 20,000 feet. White paper, August 2005.

[14] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano. The vector floating-point unit in a synergistic processor element of a Cell processor. In *Proc. 17th IEEE Symposium on Computer Arithmetic*, pages 59–67, Washington, DC, USA, 2005. IEEE (ARITH '05) Computer Society.

[15] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[16] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[17] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[18] D. Pham and *et al.* The design and implementation of a first-generation Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 184–185, San Fransisco, CA, USA, February 2005.

[19] V. Ramachandran. A general-purpose shared-memory model for parallel computation. In M. T. Heath, A. Ranade, and R. S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105, pages 1–18. Springer-Verlag, New York, 1999.

[20] M. Reid-Miller. List ranking and list scan on the Cray C-90. *J. Comput. Syst. Sci.*, 53(3):344–356, December 1996.

[21] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[22] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proc. 3rd Conference on Computing Frontiers (CF '06)*, pages 9–20, New York, NY, USA, 2006. ACM Press.