Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 33 (2007) 720–740
PARALLEL COMPUTING
www.elsevier.com/locate/parco

High performance combinatorial algorithm design on the Cell Broadband Engine processor

David A. Bader *, Virat Agarwal, Kamesh Madduri, Seunghwa Kang

Georgia Institute of Technology, Atlanta, GA 30332, United States

Received 25 April 2007; received in revised form 9 September 2007; accepted 27 September 2007
Available online 5 October 2007

Abstract

The Sony–Toshiba–IBM Cell Broadband Engine (Cell/B.E.) is a heterogeneous multicore architecture that consists of a traditional microprocessor (PPE) with eight SIMD co-processing units (SPEs) integrated on-chip. While the Cell/B.E. processor is architected for multimedia applications with regular processing requirements, we are interested in its performance on problems with non-uniform memory access patterns. In this article, we present two case studies to illustrate the design and implementation of parallel combinatorial algorithms on Cell/B.E.: we discuss list ranking, a fundamental kernel for graph problems, and zlib, a data compression and decompression library.

List ranking is a particularly challenging problem to parallelize on current cache-based and distributed memory architectures due to its low computational intensity and irregular memory access patterns. To tolerate memory latency on the Cell/B.E. processor, we decompose work into several independent tasks and coordinate computation using the novel idea of *Software-Managed threads* (SM-Threads). We apply this generic SPE work-partitioning technique to efficiently implement list ranking, and demonstrate substantial speedup in comparison to traditional cache-based microprocessors. For instance, on a 3.2 GHz IBM QS20 Cell/B.E. blade, for a random linked list of 1 million nodes, we achieve an overall speedup of 8.34 over a PPE-only implementation.

Our second case study, zlib, is a data compression/decompression library that is extensively used in both scientific as well as general purpose computing. The core kernels in the zlib library are the LZ77 longest subsequence matching algorithm and Huffman data encoding. We design efficient parallel algorithms for these combinatorial kernels, and exploit concurrency at multiple levels on the Cell/B.E. processor. We also present a Cell/B.E. optimized implementation of gzip, a popular file-compression application based on the zlib library. For our Cell/B.E. implementation of gzip, we achieve an average speedup of 2.9 in compression over current workstations.

Keywords: Combinatorial algorithms; Cell Broadband Engine processor; Multicore; List ranking; zlib; Parallel algorithms; Graph algorithms; Performance; Novel architectures

* Corresponding author.
E-mail addresses: bader@cc.gatech.edu (D.A. Bader), virat@cc.gatech.edu (V. Agarwal), kamesh@cc.gatech.edu (K. Madduri), s.kang@gatech.edu (S. Kang).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2007.09.005

1. Introduction

The Cell Broadband Engine (or the Cell/B.E.) [17] 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 multiprocessor 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. There are several unique architectural features in Cell/B.E. that clearly distinguish it from current microprocessors: the Cell/B.E. chip is a computational workhorse, and it offers a theoretical peak single-precision floating-point performance of 204.8 GFlop/s. We can exploit parallelism at multiple levels on the Cell/B.E.: 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]. The on-chip interconnection network elements have been specially designed to cater for high performance on bandwidth-intensive applications (such as those in gaming and multimedia).

All these features make the Cell Broadband Engine attractive for scientific computing, as well as an alternative architecture for general purpose computing. Williams et al. [25] analyzed the performance of Cell/B.E. 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/B.E. performs impressively for applications with predictable memory access patterns, and that communication and computation can be overlapped more effectively on the Cell/B.E. than on conventional cache-based approaches. Noting that while Cell/B.E. is architected for multimedia applications with regular processing requirements, we are interested in its performance on problems with predominantly integer-based computations and non-uniform memory access patterns. We challenge the general perception that the Cell/B.E. architecture is not suited for problems that involve fine-grained memory accesses, and where there is insufficient computation to hide memory latency. In this article, we present two representative case studies from the class of combinatorial computing problems, the list ranking kernel and zlib data compression and decompression library, for which we achieve impressive performance on the Cell/B.E. processor.

List ranking [7,23,15] is a fundamental paradigm for the design of many parallel combinatorial and graph-theoretic applications. 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. In prior work, using list ranking as a sub-routine, we 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, tree contraction and expression evaluation [4], spanning tree [1] and minimum spanning forest [2]. Due to its low computational intensity and lack of locality, it is difficult to attain parallel speedup for this problem. By applying our new techniques for latency tolerance and load balancing with *Software-Managed threads* (SM-Threads), our list ranking implementation on the Cell/B.E. processor achieves significant speedup. We conduct an extensive experimental study comparing our Cell/B.E. 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 Cell Broadband Engine performance by a factor of about 4.1 for both random and ordered lists.
- By tuning just one algorithm parameter, our list ranking implementation is load-balanced across the SPEs with high probability, even for random lists.
- The Cell/B.E. achieves an average speedup of eight 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.

We discuss the list ranking case study in Section 4, and introduce SM-threads and our latency-hiding technique in Section 4.3.

Data compression and decompression applications are extensively used in scientific and general purpose computing, and the open-source zlib package [9] is a popular library for compression/decompression. zlib is based on the LZ77 subsequence matching algorithm: data is compressed by first identifying repeating subsequences of characters, and then replacing duplicate strings with reference to their previous occurrences. The strings and references are further Huffman-coded to improve compression. The performance of the compression algorithm is data-dependent, while the decompression algorithm involves a high percentage of table lookups for finding matches. We exploit concurrency at multiple levels for both compression and decompression. Since the SPEs support SIMD instructions, we identify compute-intensive routines and vectorize them. At a higher level, we also partition the computation among various SPEs. For performance evaluation, we design a Cell/B.E. optimized implementation of gzip, a popular file-compression application based on the zlib library. We use the full flushing technique supported by gzip to partition a data file across multiple SPEs, and achieve an average speedup of five over current cache-based microprocessors. We present a detailed discussion of our design and implementation of the zlib library in Section 5.

The heterogeneous processors, limited on-chip memory and multiple avenues for parallelism on the Cell/B.E. processor make algorithm design and implementation a new challenge, with potentially high payoffs in terms of performance. Analyzing algorithms using traditional sequential complexity models like the RAM model fail to account for several Cell/B.E. architectural intricacies. There is currently no simple and accepted model of computation for the Cell/B.E., and in general, for multicore architectures. We use a simple complexity model for the design and analysis of parallel algorithms on the Cell Broadband Engine architecture. We express the algorithm complexity on the Cell/B.E. 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 3. We then present a *systematic methodology* for analyzing algorithms using this complexity model, and illustrate this with an example of matrix multiplication.

2. Cell Broadband Engine architecture

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multicore chip that is significantly different from conventional multiprocessor 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. Fig. 1 gives an architectural overview of the Cell/B.E. We refer the reader to [21,10,18] for additional details.

The PPE is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KB L1 instruction and data caches, and a 512 KB L2 cache. It is a dual issue, in-order execution design, with two-way simultaneous multithreading. Ideally, all the computation should be partitioned among the SPEs with PPE handling the control flow.

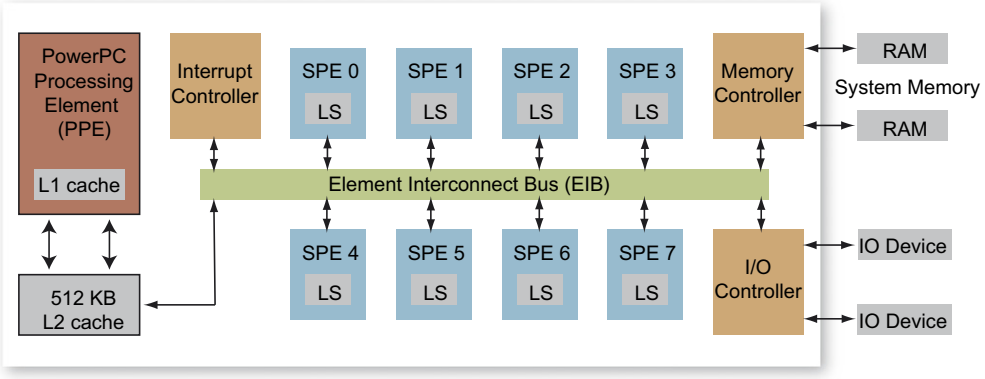

Fig. 1. Cell Broadband Engine architecture.

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 SPU is a micro-architecture designed for high performance data streaming and data intensive computation. It includes a 256 KB *local store* (LS) memory to hold SPU program's instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into the local store or write computation results back to the main memory. DMA is non-blocking so that the SPU can continue program execution while DMA transactions are performed.

The SPU is an in-order dual-issue statically scheduled architecture. Two SIMD [14] instructions can be issued per cycle: one compute instruction and one memory operation. The SPU branch architecture does not include dynamic branch prediction, but instead relies on compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as much as possible.

The MFC supports naturally aligned transfers of 1,2,4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 KB. DMA list commands can request a list of up to 2048 DMA transfers using a single MFC DMA command. Peak performance is achieved when both the effective address and the local storage address are 128 bytes aligned and the transfer is an even multiple of 128 bytes. In the Cell/B.E. processor, each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in communication. Kistler et al. [18] analyze the communication network of the Cell/B.E. 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.

With a clock speed of 3.2 GHz, the Cell/B.E. processor has a theoretical peak performance of 204.8 GFlop/s (single precision). The EIB supports a peak bandwidth of 204.8 GB/s for intrachip transfers among the SPEs. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

3. Algorithm design and analysis

3.1. A complexity model for Cell/B.E.

There are several architectural features of the Cell/B.E. processor 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 support 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/B.E.. 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 $1 \leqslant i \leqslant p$, 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. Sim-

ilarly, 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 [8], BSP [24] and QSM [22]. 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 [25].

3.2. A procedure for algorithm analysis

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

(1) We compute the computational complexity $T_{\mathrm{C,SPE}}$.
(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 $1 \leqslant i, j \leqslant n$) on the Cell/B.E. is analyzed as follows:

- We partition computation among the eight SPEs by assigning each SPE the Matrix B and $\frac{n}{p}$ rows of Matrix A.
- 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 $\mathrm{O}\left(\frac{n^2}{b^2}\right)$ DMA transfers, and T_{D} is $\mathrm{O}\left(\frac{n^2}{pb^2}\right)$.

Chen et al. describe their Cell/B.E. implementation of this algorithm in [6]. The algorithmic complexity is given by $T_{\mathrm{C}} = \mathrm{O}\left(\frac{n^3}{p}\right)$, $T_{\mathrm{D}} = \mathrm{O}\left(\frac{n^2}{pb^2}\right)$ and $T_{\mathrm{B}} = \mathrm{O}(n^2)$. 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 $\mathrm{O}(nb^2)$ (as the complexity of computing b^2 elements in C is $\mathrm{O}(n)$). 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 $\mathrm{O}(n^2)$ branches due to the absence of a branch predictor on the SPE, which might degrade performance if they result in mispredicts. Using SIMD and dual-issue features within the SPE, 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/B.E.

4. List ranking using the Cell Broadband Engine

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 (see Fig. 2). 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.

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 $X(i) \cdot value$ denote its value and $X(i) \cdot next$ the index of its successor. Then for any binary associative operator $\oplus$, compute $X(i) \cdot prefix$ such that $X(head) \cdot prefix = X(head) \cdot value$ and $X(i) \cdot prefix = X(i) \cdot value \oplus X(predecessor) \cdot prefix$, 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 the Cell/B.E. and analyze it as follows. Assume that we start with eight sublists, one per SPE. Using DMA fetches, the SPEs continue obtaining the successor elements until they reach a sublist end, or the end of the list.

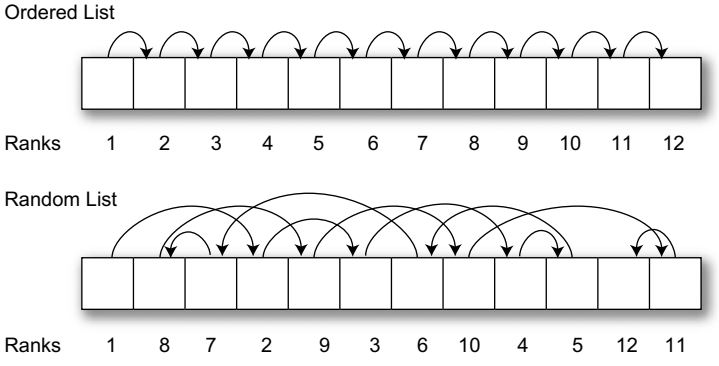

Fig. 2. List ranking for ordered (top) and random (bottom) list.

4.2. Applying the complexity model to list ranking

Analyzing the complexity of this algorithm using our model, we have $T_C = O\left(\frac{n}{p}\right)$, $T_D = O\left(\frac{n}{p}\right)$ 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.

4.3. 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 the Cell/B.E.

Since the Cell/B.E. 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 does 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 architectures such as the Cray MTA-2 and NVIDIA G80 that use massive multithreading to hide latency. 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 per SPE is active. We keep switching software contexts so that we do computation between a DMA request and its completion. We use a round-robin schedule for the threads (see Fig. 3).

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 incurs a significant

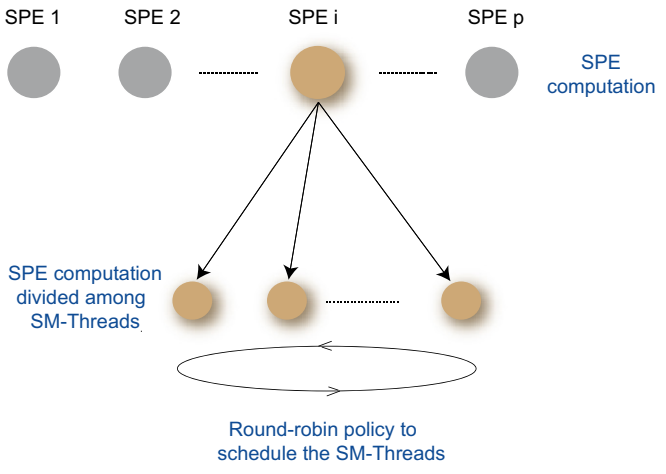

Fig. 3. Illustration of the technique.

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 the Cell/B.E.

4.4. Implementation

Our Cell/B.E. 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. Fig. 4 illustrates Step 3 of this algorithm.
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 Section 4.5) 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

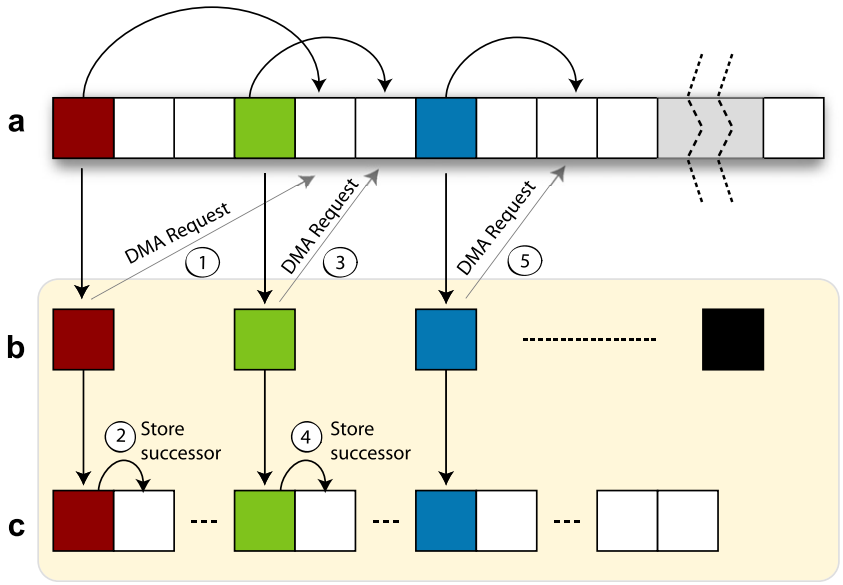

Fig. 4. Step 2 of list ranking on the Cell/B.E. (a) Linked list for which list ranking is to be done. Highlighted nodes in this figure are allocated to SPE(i); (b) View from SPE(i), it has s/p sublist head nodes to traverse concurrently; (c) This array is used to store sublists in contiguous area of memory. When this gets full, we transfer it back to the main memory.

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 $\alpha(s) \geqslant 2.62$. Thus, the load is balanced among various SPEs under this constraint.

In our implementation, we incorporate recommended software strategies [5] and techniques to exploit the architectural features of the Cell/B.E. For instance, we use manual loop unrolling, branch hints, and design our implementation for a limited local store.

4.5. Performance results

We report our performance results from actual runs on a IBM BladeCenter QS20, with two 3.2 GHz Cell/B.E. 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 2.0, 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 in this paper is a generic work partitioning technique to hide memory latency. We demonstrate the results of this technique for list ranking: we use SM-threads 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. 5 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 (90 ns, 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 eight DMA buffers.

In Fig. 6, we present the results for load balancing among the eight 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. 6 illustrates this: load balancing is better for 64 sublists than the case of eight sublists.

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

Fig. 7 shows the running time of our Cell/B.E. 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.
Intel_WC:	2.67 GHz Intel Dual-Core Xeon 5150 processor (Woodcrest), Intel C compiler v9.1.
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.

For Intel Xeon 5150 (Woodcrest) we use a parallel implementation [3] running on two threads (see Table 1). Table 2 gives the comparison of running time of various steps in the list ranking algorithm. The table shows that Step 2 dominates the entire running time of the algorithm. Finally, we demonstrate a substantial speedup of our Cell/B.E. implementation over a sequential implementation using the PPE-only. We compare the best

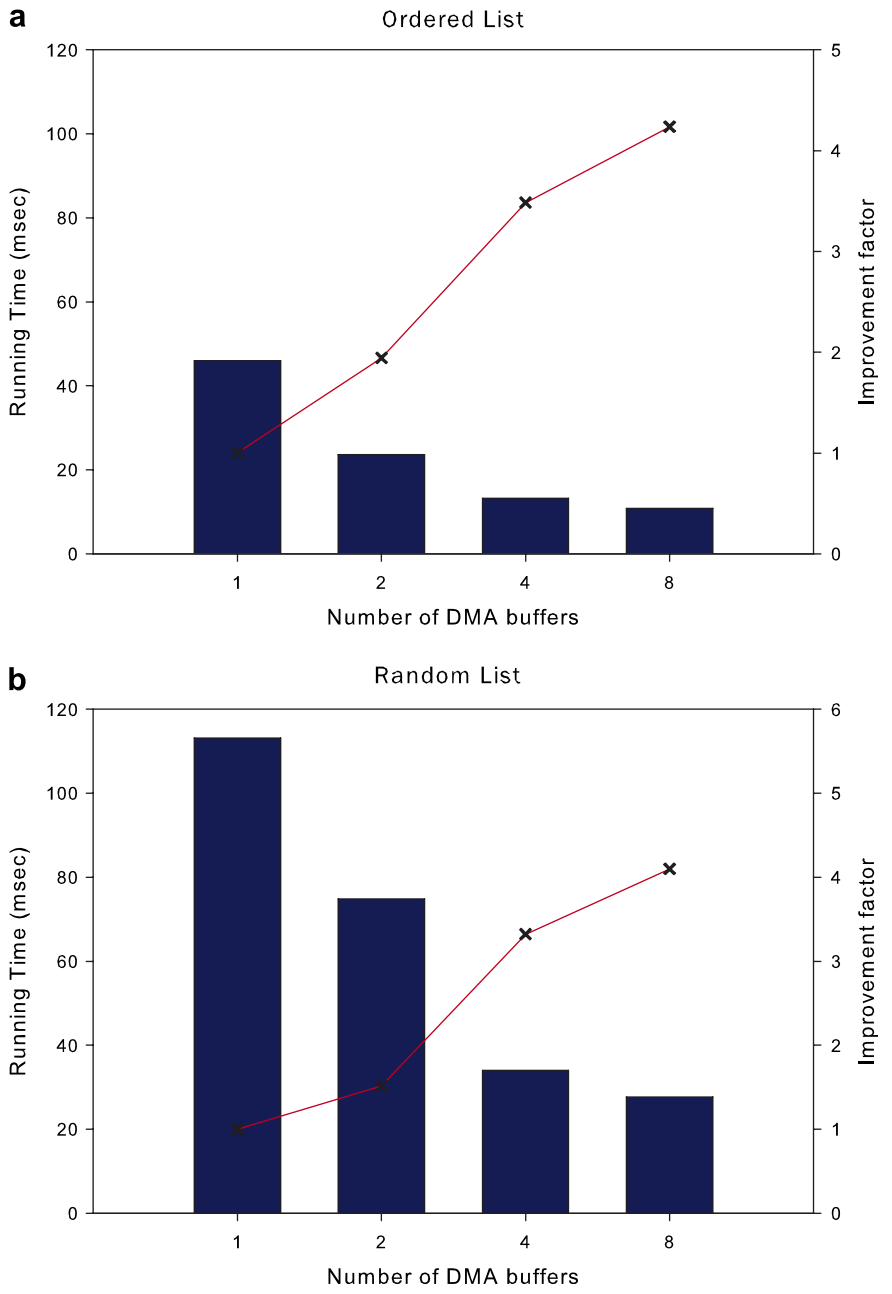

Fig. 5. Achieving latency tolerance through DMA parameter tuning for ordered (top) and random (bottom) list of size 2^{20}.

sequential approach that uses pointer-chasing to our algorithm using different problem instances. Fig. 8 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. Data compression/decompression on the Cell Broadband Engine

We now discuss the next case study of optimizing compression/decompression on the Cell/B.E. processor using zlib [9].

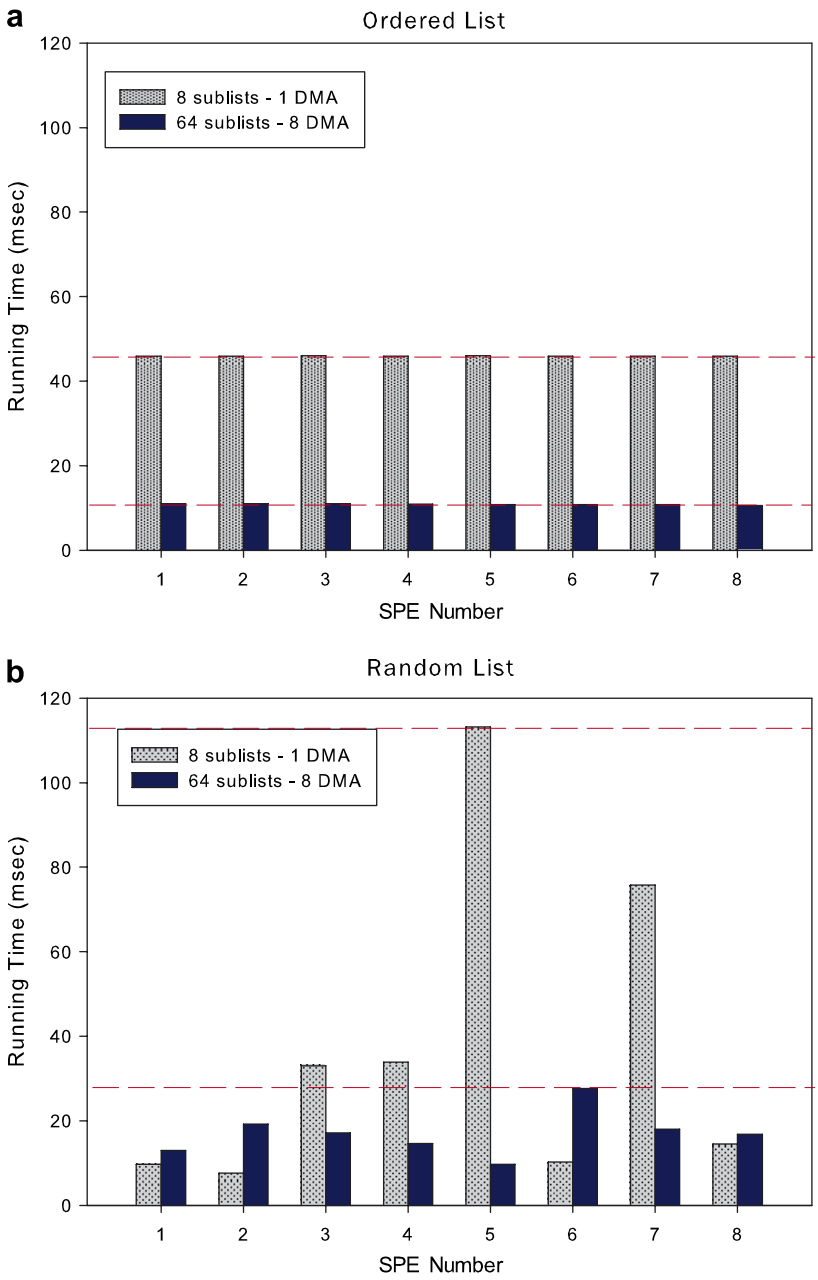

Fig. 6. Load balancing among SPEs for ordered (top) and random (bottom) for list ranking on the Cell/B.E. with a list of size 2^{20}.

5.1. Description of zlib algorithm

The zlib library is based on the LZ77 algorithm [26] and Huffman coding [13]. In the LZ77 algorithm, a data stream is compressed by first identifying the longest repeating string, and then replacing the duplicated string with a reference to its previous occurrence. This reference is represented by a *length–distance* pair, where *length* specifies the length of the repeated string and *distance* gives the offset of this occurrence. During compression, a hash table is constructed and used to make the process of finding matches faster. For strings that do not have a previous match, the input literal contained in the string is added to the compressed data. A further compression is achieved by Huffman coding. Huffman coding enhances the compression ratio by

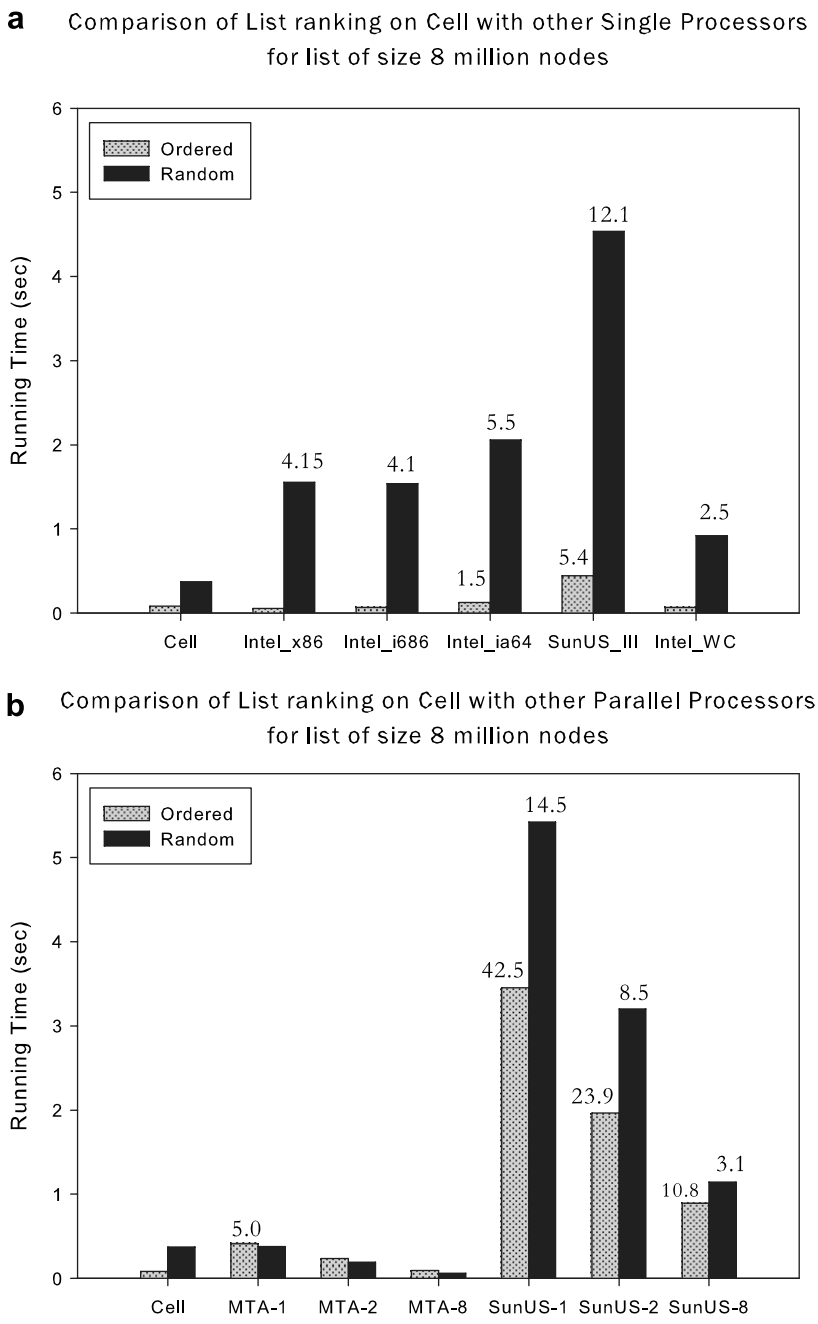

Fig. 7. Performance of list ranking on the Cell/B.E. as compared to other single processor and parallel architectures for lists of size 8 million nodes. The speedup of the Cell/B.E. implementation over the architectures is given above the respective bars.

encoding symbols with varying bit lengths according to the frequency of occurrences. In particular, shorter codes are assigned for more frequent symbols, and longer codes are assigned for less frequent symbols. During decompression, length–distance pairs and literals are retrieved from Huffman-coded data and the reference pairs are replaced to obtain original data from the compressed data.

In Fig. 9, we have an illustration of the LZ77 algorithm. Suppose the current sliding window covers literals starting with index 8 in the input sequence. The hash function generates a hash key by using the first 3 bytes (index from 8 to 10) of the input string that helps locate the most recent occurrence of this string. The algo-

Table 1
Speedup of our Cell/B.E. implementation of list ranking as compared with other architectures

Architecture	Speedup (ordered)	Speedup (random)
Intel_x86	0.85	4.15
Intel_i686	0.9	4.1
Intel_ia64	1.5	5.5
SunUS_III	5.4	12.1
Intel_WC	0.82	2.5
MTA-1	5.0	1.05
SunUS_II-1	42.5	14.5
SunUS_II-2	23.9	8.5
SunUS_II-8	10.8	3.1

Table 2
Running time comparison of the various steps of list ranking on Cell/B.E. for ordered and random lists of size 8 million nodes

Task	Ordered (%)	Random (%)
Step 1: Identification of sublist head nodes	0.09	0.02
Step 2: Traversal of sublists	99.84	99.96
Step 3: Computing the ranks	0.07	0.02

rithm also maintains a backward chain for the strings with the identical hash key value using a separate data structure. This is used during a backward traversal, during which a comparison is made with each string to identify the match length. The string that has the longest match is located and the repeated string is replaced by a reference to the located string. Also, the hash table is updated to point to this location and the backward chain is updated to start from this location.

Several input parameters in zlib result in a trade-off between *compression ratio*, *speed* and *memory requirements*. One such parameter is the *window size*, which determines the size of the recent input data to be stored for searching matches. A large window size can help achieve a better compression ratio, but results in a large working set. Another configuration option is the *hash table size*. A big hash table leads to less collisions that makes the program run faster. It also helps achieve a better compression ratio, but leads to higher memory utilization.

The compression/decompression stages in zlib are highly dependent in data processing which makes it difficult to parallelize. For example, during decompression of the Huffman-coded data, the stream of input signals needs to be decoded sequentially. This is because the symbols have varying bit lengths and we do not know where the next symbol starts before its previous symbol is decoded. Also, Huffman-coded data is a mix of literals and length–distance pairs. In order to determine if the next symbol is a literal, length, or distance, the previous symbol must be decoded.

Another design issue is that the zlib algorithm requires a substantial number of table lookups. For example, finding matches and decoding the Huffman-coded data stream require accessing the hash and code tables, respectively. Since the SPE does not support scatter and gather type memory access to the local store, this cannot be vectorized. In addition, zlib algorithm has many branches which are dependent on input data and hard to predict statically. For a stream with poor compression ratio, almost every symbol is a literal. On the other hand, for highly compressible streams, most symbols are length–distance pairs. This leads to different branch behaviors and due to the absence of a dynamic branch predictor in the SPEs, these branch mispredictions reduce the performance.

5.2. Applying the complexity model to zlib

We apply the complexity model discussed in Section 3.1 to analyze zlib algorithm. The computational complexity of the sequential algorithm is $O(n)$, as we make one pass over the data stream and the hash table lookups are constant-time. In our parallel implementation of zlib, we partition the input data into multiple blocks.

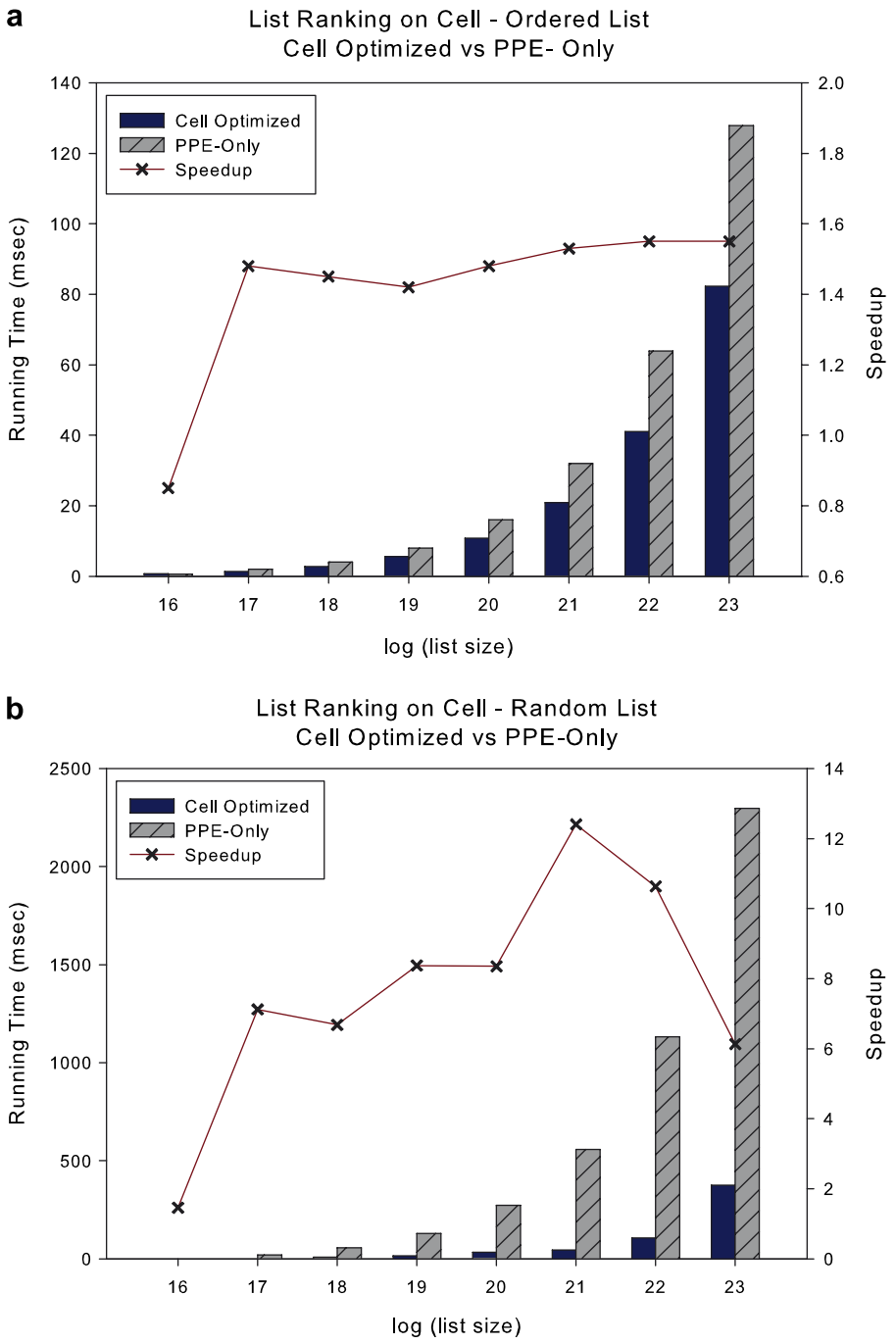

Fig. 8. Performance comparison of sequential implementation on PPE to our parallel implementation of list ranking on the Cell/B.E. for ordered (top) and random (bottom) lists.

We use the work-queue strategy detailed in Section 5.4 for load-balanced computation. Thus, $T_{C,SPE}$ becomes $O\left(\frac{n}{p}\right)$.

Let us assume that the DMA transfer block size is b. T_D, the complexity of DMA requests, is given by $O(n/pb)$. Note that in zlib the DMA transfer block size is larger than that of list ranking, where we use a block size of O(1). From Step 2 of the algorithm analysis procedure, since the complexity of DMA fetches is a

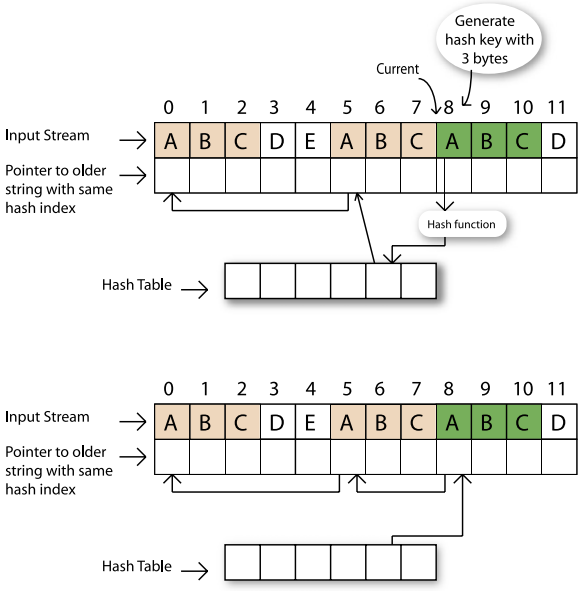

Fig. 9. An illustration of the LZ77 algorithm.

function of n, we analyze the computational complexity in the average case between a DMA request and its completion. For zlib this is given by O(b). Since this is not a function of the input parameters, further analysis is required to determine the amount of data buffering required.

The zlib algorithm is branch intensive, and branch instructions on the Cell/B.E. are hard to predict statically. T_B, the number of branch instructions, is O(n). However, we can still achieve speed-up by parallelization, vectorization and other Cell/B.E. processor specific optimizations, which suggests that the Cell/B.E. processor can be used for wider range of applications.

5.3. Optimizing zlib for the Cell Broadband Engine

The first step towards optimization of an algorithm on the Cell/B.E. involves the identification of the compute-intensive parts. Table 3 shows the result of profiling for the compression routine in zlib.

During compression, a new string is inserted into the hash table whenever a match is found. This step also consists of a intensive loop that iterates over for the length of the match and inserts a single byte into the hash table in each iteration. The hash key is computed using the 3 bytes starting from the inserting byte, with an overlap of 2 bytes that were considered during the previous iteration. In the original implementation, this overlap is exploited and the hash key is computed from the previous key to reduce the amount of computation. This gives performance benefit on a scalar machine. However, this generates a false dependency which makes it tough to vectorize for the Cell/B.E. processor. We break this dependency by calculating the hash key using all 3 bytes in every iteration without using the previous hash key. This generates redundancy in computation but improves performance in overall by vectorization.

Table 3
Percentage of execution time of the various tasks during compression

Task	Execution time (%)
Fill window for previous data to find matches	11.73
Find the longest matches in LZ77 algorithm	32.69
Update the output buffers and statistics according to the availability of the matches	36.89
Perform Huffman coding on the data	11.81

Another step in compression requires finding the longest match where a comparison is made between the current string and the previous strings. This step consists of intensive loops with byte comparisons, which can be implemented using the 16 byte byte-wise vector comparison instruction. This optimization leads to a significant speed up if data has high redundancy, whereas for a less redundant input data comparison is made with only small number of bytes which reduces the performance benefit.

In the compression process, zlib manages a sliding window for finding the longest match using the LZ77 algorithm. The window is divided into two parts, the first half stores the previous data stream that is used to find the matches and the second half stores the current stream that needs to be compressed. When the current pointer reaches to the end of the sliding window, the second half of the sliding window is copied to the first half, and a new data stream is loaded to the second half. Accordingly, the hash table and backward chain, which have the pointers to the sliding window, need to be updated. This is a intensive loop and the iteration count of the loop depends on window and hash table size. We vectorized this loop and manually unroll it to achieve better pipeline utilization. The pointer variable in this loop is 2 bytes long, thus vectorization reduces the loop iterations by a factor of eight.

Table 4 shows the result of profiling for the decompression routine in zlib. The most compute-intensive task during decompression is first, converting the Huffman-coded data to literals or length-distance pairs and then, replacing the references with the original data. The first step, is highly data-dependent, which makes it hard to vectorize. The second step is essentially a byte-wise memory copy that can be vectorized. We can copy 16 bytes at once using vector instructions. However, if there is an overlap between the source and destination vectors this technique does not work (Fig. 10). To overcome this problem we shuffle the source vector using a table lookup for shuffle pattern, and copy the generated output vector to the destination.

Another computation intensive step during decompression involves the calculation of CRC values. CRC algorithms for SIMD architectures have been studied by Joshi et al. [16] and Lin [19]. Lin [19] introduced the concept of 'congruent equivalence', and replaced table lookups with the shuffle instructions. Table lookups

Table 4
Percentage of execution time of the various tasks during decompression

Task	Execution time (%)
Convert Huffman-coded data to literal, and length–distance pairs, and copy data from previous input according to these pairs	77.25
Construct table for decompression	8.72
Calculate the CRC value	7.96
Parse header data	5.97

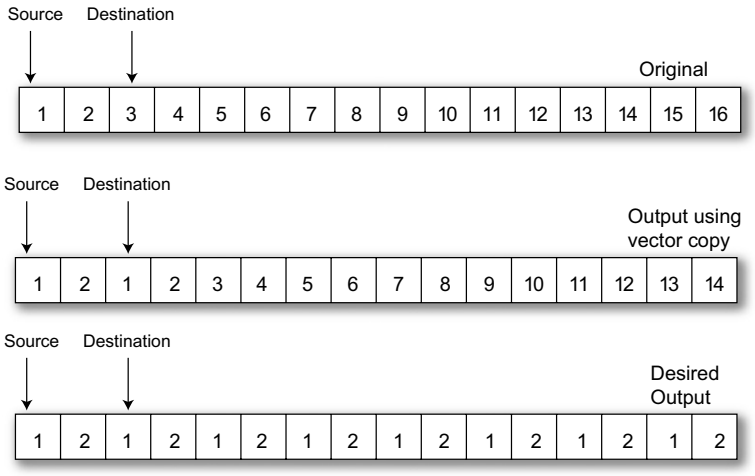

Fig. 10. Illustration of problem in byte-wise memory copy using vector instruction.

can be replaced with the shuffle instructions if table size is equal to or smaller than 32 bytes. Joshi et al. and Lin used nibble-wise table lookup instead of traditional byte-wise lookup to reduce the table size. These approaches are attractive for the Cell/B.E. processor as the SPE supports the shuffle instruction. We also vectorize the Adler32 algorithm [20], which can be used instead of the crc32( ) function.

We use statistical approaches for branch hinting and loop optimization. *Loop unrolling* works for intensive loops, but adds overhead and increases code size. This does not help for loops with a small number of iterations. So we identify the compute-intensive loops and apply optimization techniques on them. This gives a performance boost and saves local store space. *Branch hint* instruction benefits especially when the branch offset is large, and branch is highly predictable statically. When compiler encounters branch hint intrinsic, compiler inserts branch hint instruction which facilitates prefetching the instruction and also make hinted case as the fall through case if possible. We use branch hint intrinsics wherever possible to reduce the number of branch mispredictions. *Reducing memory usage* is also important in the optimization. In a virtual memory system, unused function code resides on disk and never loaded to the main memory or the cache. In the SPE, the entire executable image is loaded to the local store. Considering that the size of local store is only 256 KB, this is undesirable. We separate compression and decompression routines to reduce code size.

5.4. Optimizing gzip on the Cell Broadband Engine

Gzip is an application that uses the zlib library for file-compression and decompression. The zlib algorithms are highly data-dependent which makes it hard to partition the input data for parallel processing. We use *full flushing* (supported by zlib API) to break this data dependency. We can then restart the algorithm after the full flush point, making the algorithm data-dependent only between two consecutive flush points. This helps in partitioning the data and enables parallel processing. However, during decompression a complete search of the compressed file is required to find the flush points $(0, 0, 0\text{xff}, 0\text{xff})$. This is an additional overhead particularly for the files compressed without full flushing. By extending gzip header format we can avoid this search overhead keeping the file compatible with the legacy gzip decompressors.

We use an extra field to indicate that the file has multiple blocks with full flush between blocks. The size of the each block is also included in the header in order to avoid full search to find synchronization points. During the decompression stage, the decompressor inspects the flags. If the sub-field ID for Cell/B.E. gzip extension is found on systems with the Cell/B.E. zlib, it decompresses the file in parallel using multiple SPEs. If not, it sequentially decompresses the file as a normal gzip file. A legacy decompressor would ignore the extra field with unknown sub-field ID and skip the bytes. Therefore, the file can still be decompressed correctly. Full flushing and additional information will slightly increase the file size by a negligible amount (0.2% increase for 900 KB block size).

The new field is structured as follows:

- Byte 1 stores the character 'C' which stands for Cell /B.E.
- Byte 2 stores the character 'E' which stands for Extension.
- We use 2 bytes for storing n, number of blocks.
- We use 4 bytes for storing the size of each block, and this requires $4n$ bytes.

We also consider load balancing and file I/O delay in our parallel implementation. It is important to note that dividing input file into chunks of equal size will not distribute work uniformly among the processors, as the compression time is highly dependent on the input file characteristics. We achieve load balancing by partitioning input file to multiple blocks. Each SPE is initially assigned with a work-task, and on completion it picks up another work-task from the work queue. We also create two additional threads for file read and write. The file read thread reads the data to be processed in advance and fill the memory buffer. The write thread writes the computed result in background.

5.5. gzip performance results

The gzip code is compiled with gcc provided with the Cell SDK 2.0 using optimization level 5 ($-$O5). We used the Cell SDK simulator for measuring clock cycles, and the IBM QS20 Cell/B.E. blade for running time

Table 5
Parameters used for compiling the zlib library

zlib library	windowBits	memLevel
Basic	13	6
Compression	13	7
Decompression	15	–

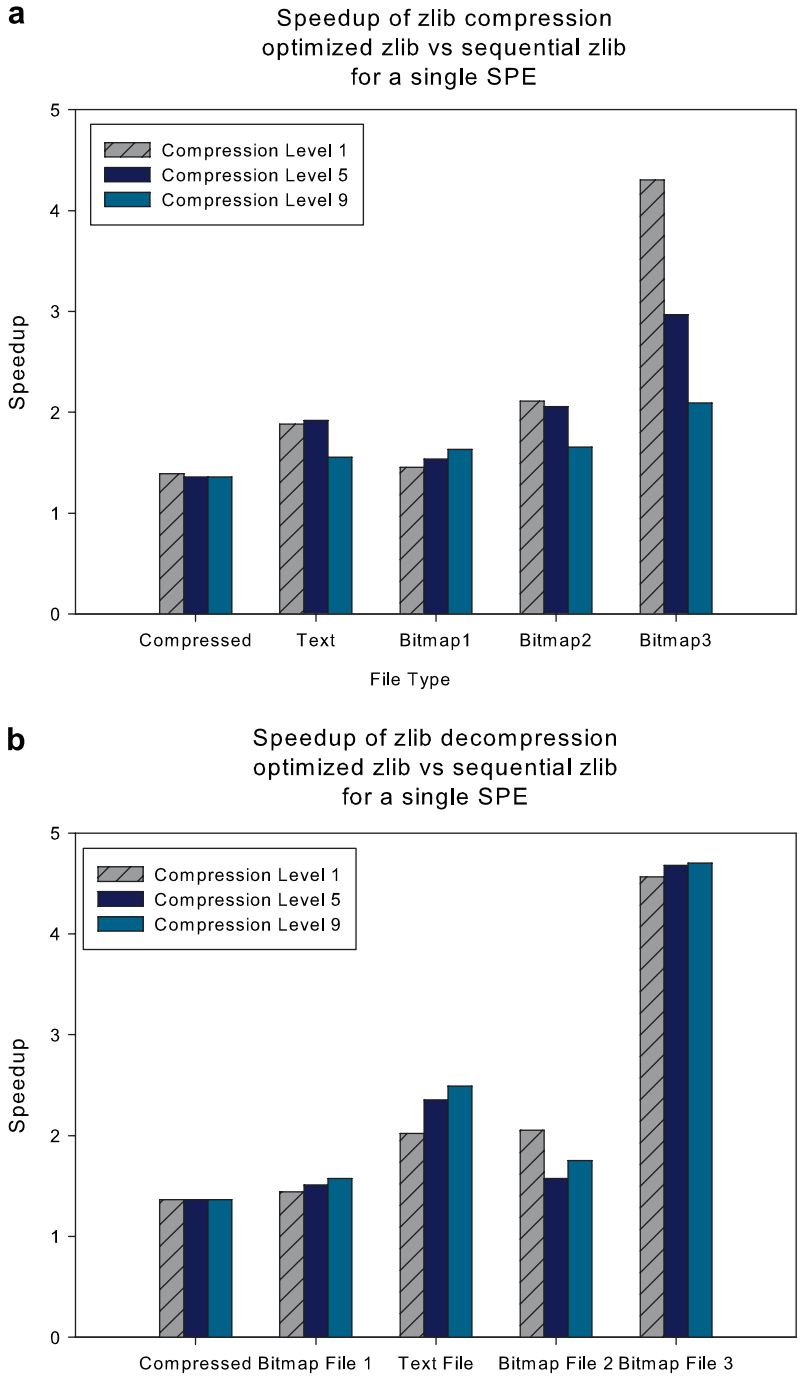

Fig. 11. Speedup of Cell/B.E. optimized gzip application on a single SPE for compression (top) and decompression (bottom).

measurements. The parameters used to compile the original zlib library, Cell/B.E. optimized compression, and decompression, are given in Table 5. The total memory use is given by

- *Compression*: $2^{windowBits+2} + 2^{memLevel+9}$ bytes.
- *Decompression*: $2^{windowBits+2}$ bytes.

Fig. 11 shows the speedup obtained for different file types using our optimized SPE implementation as compared to the gzip program using original zlib library run on the single SPE. The speedup is achieved by SIMDization and other Cell/B.E. specific optimizations including branch hints and loop unrolling. The test files we have used for performance analysis consist of an already compressed file that has very less redundancy, Bitmap file 1, 2 and 3 that have increasing level of redundancy, with Bitmap file 3 having 100% redundancy. Please refer to Table 6 for the compression ratio of these file types. Fig. 11 shows that the speedup obtained, during compression and decompression stages, is highly dependent on the input file type where we get performance improvement of over 4 for Bitmap file 3 and an improvement of 1.5 for an already compressed file.

Fig. 12 shows the performance comparison of the Cell/B.E. optimized gzip compression as compared with other single processor architectures. Please refer to Section 4.5 for details of these architectures. For Intel Xeon 5150 (Woodcrest) we ran gzip using a single thread. Due to the unavailability of a opensource parallel gzip application our performance is limited to single processor architectures.

Table 6
Compression ratio $\left(\frac{\text{compressed file size}}{\text{original file size}} \times 100\right)$ using gzip for the various test files

Test file	Compression level 1 (%)	Compression level 5 (%)	Compression level 9 (%)
Compressed file (484 KB)	100	100	100
Bitmap file (355 KB)	35.8	31.0	29.4
Text file (460 KB)	11.3	8.16	7.48
Bitmap file 2 (546 KB)	7.84	7.35	7.08
Bitmap file 3 (2.40 MB)	0.499	0.158	0.138

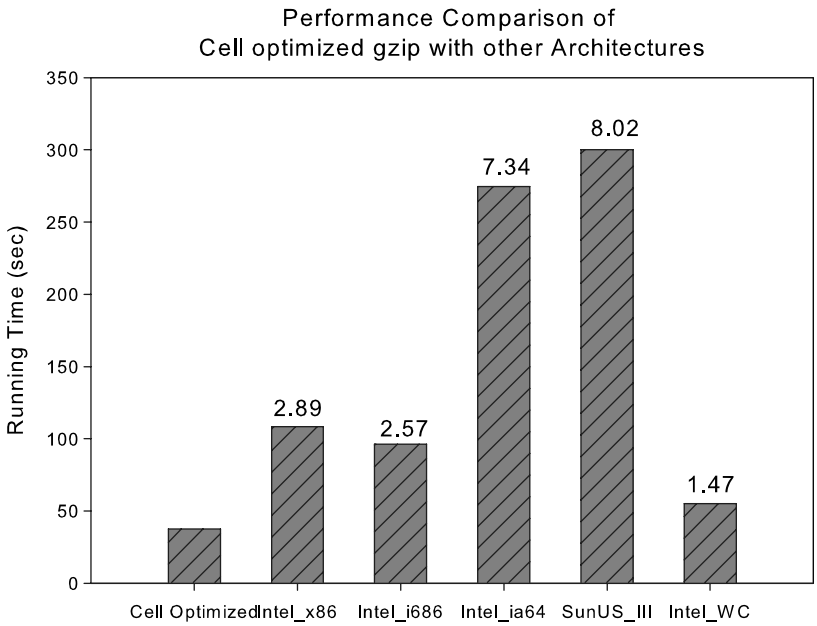

Fig. 12. Performance comparison of Cell/B.E. optimized gzip compression with the original zlib implementation on other single processor architectures.

6. Conclusions

In summary, we present two case studies to illustrate the design and implementation of parallel combinatorial algorithms on the Cell/B.E. processor: list ranking, a fundamental kernel in graph algorithms, and zlib, a data compression and decompression library. We introduce the idea of Software-Managed threads (SM-Threads), a generic work-partitioning technique for latency tolerance and load balancing on the Cell/B.E. We apply this technique to develop a fast parallel implementation of list ranking, and confirm its efficacy by demonstrating an improvement factor of 4.1 as we tune relevant parameters. Most importantly, we achieve an overall speedup of 8.34 of our implementation over an efficient PPE-only sequential implementation. Similarly, for the zlib library, we exploit concurrency at multiple levels: we vectorize compute-intensive kernels in compression and decompression, and also parallelize the functions to run on multiple SPEs. Our Cell/B.E. implementation of the gzip application, based on the zlib library, achieves a speedup of 2.9 for compression over a high-end Intel Pentium-4 system. With these case studies, we highlight the potential of the Cell/B.E. processor as an accelerator for combinatorial applications.

Acknowledgements

This work was supported in part by NSF Grants CNS-0614915, CAREER CCF-0611589, DBI-0420513, ITR EF/BIO 03-31654, and an IBM Shared University Research (SUR) Grant. We would also like to thank Sidney Manning (IBM Corporation) and Vipin Sachdeva (IBM Research) for providing valuable inputs during the course of our research. We also acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

[1] D.A. Bader, G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs), in: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, NM, April 2004.
[2] D.A. Bader, G. Cong, Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs, in: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, NM, April 2004.
[3] D.A. Bader, G. Cong, J. Feo, On the architectural requirements for efficient execution of graph algorithms, in: Proceedings of the 34th International Conference on Parallel Processing (ICPP), Oslo, Norway, June 2005.
[4] D.A. Bader, S. Sreshta, 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, U. Shukla (Eds.), Proceedings of the Ninth International Conference on High Performance Computing (HiPC 2002), Bangalore, India, Lecture Notes in Computer Science, vol. 2552, Springer-Verlag, 2002, pp. 63–75.
[5] D.A. Brokenshire, Maximizing the power of the Cell Broadband Engine Processor: 25 tips to optimal application performance, IBM developerWorks technical article, 2006.
[6] T. Chen, R. Raghavan, J. Dale, E. Iwata, Cell Broadband Engine Architecture and its first implementation, IBM developerWorks technical article, 2005.
[7] R. Cole, U. Vishkin, Faster optimal prefix sums and list ranking, Information and Computation 81 (3) (1989) 344–352.
[8] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation, in: Proceedings of the Fourth Symposium Principles and Practice of Parallel Programming, ACM SIGPLAN, May 1993, pp. 1–12.
[9] P. Deutsch, J.-L. Gailly, zlib compressed data format specification version 3.3, Internet RFCs, 1996.
[10] B. Flachs et al., A streaming processor unit for a Cell processor, in: Proceedings of the International Solid State Circuits Conference, vol. 1, San Fransisco, CA, USA, February 2005, pp. 134–135.
[11] D.R. Helman, J. JáJá, Designing practical efficient algorithms for symmetric multiprocessors, in: Algorithm Engineering and Experimentation (ALENEX'99), Baltimore, MD, Lecture Notes in Computer Science, vol. 1619, Springer-Verlag, 1999, pp. 37–56.
[12] D.R. Helman, J. JáJá, Prefix computations on symmetric multiprocessors, Journal of Parallel and Distributed Computing 61 (2) (2001) 265–278.
[13] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE 40 (9) (1952) 1098–1101.
[14] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, The vector floating-point unit in a synergistic processor element of a Cell processor, in: Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Washington, DC, USA, 2005, IEEE (ARITH'05) Computer Society, pp. 59–67.
[15] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley Publishing Company, New York, 1992.

[16] S.M. Joshi, P.K. Dubey, M.A. Kaplan, A new parallel algorithm for CRC generation, IEEE International Conference on Communications 3 (2000) 1764–1768.
[17] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, D. Shippy, Introduction to the Cell multiprocessor, IBM Journal of Research Development 49 (4/5) (2005) 589–604.
[18] M. Kistler, M. Perrone, F. Petrini, Cell multiprocessor communication network: built for speed, IEEE Micro 26 (3) (2006) 10–23.
[19] B. Lin, Altivec solutions to sequential problems: calculating CRC with scalable congruent equivalent compression, Freescale application note AN2926, January 2006.
[20] K. Margaritis, Vectorization of algorithm Adler32 using altivec. `freevec.org` whitepaper, 2005.
[21] D. Pham et al. The design and implementation of a first-generation Cell processor, in: Proceedings of the International Solid State Circuits Conference, vol. 1, San Fransisco, CA, USA, February 2005, pp. 184–185.
[22] V. Ramachandran, A general-purpose shared-memory model for parallel computation, in: M.T. Heath, A. Ranade, R.S. Schreiber (Eds.), Algorithms for Parallel Processing, vol. 105, Springer-Verlag, New York, 1999, pp. 1–18.
[23] M. Reid-Miller, List ranking and list scan on the Cray C-90, Journal of Computer and System Sciences 53 (3) (1996) 344–356.
[24] L.G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.
[25] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, The potential of the Cell processor for scientific computing, in: Proceedings of the Third Conference on Computing Frontiers CF'06, 2006, ACM Press, New York, NY, USA, pp. 9–20.
[26] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.