-

Radix sort on the Cell Broadband Engine

N. Ramprasad and Pallav Kumar Baruah
Department of Mathematics and Computer Science,
Sri Sathya Sai University

Abstract:

Radix sort algorithm does not perform comparisons between input elements. Due to its enormous data parallelism it is more suitable for sorting data in parallel. In this work we suggest an optimization for the parallel radix sort algorithm, which reduces the time complexity of the algorithm and also ensures that the load on all the processors is balanced. We have implemented it on the "Cell processor" which the first implementation of the Cell Broadband Engine Architecture (CBEA) designed by the STI (Sony, Toshiba and IBM). It is a heterogeneous multi-core processor system. We have been able to sort 102400000 elements in 0.49 seconds at a rate of 207 Million/sec.

1. Introduction

a. Radix sort

Radix sort is a multiple-pass algorithm, which distributes each item to a *bucket* according to part of the item's *key,* beginning with the least significant part of the key. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key.

Here is an example of radix sort algorithm. Let {164, 37, 395, 12, 9, 982, 733, 112, 428, 377, and 55} be the numbers to be sorted in the ascending order.

Initially	Pass 1	Pass 2	Finally
1 6 4	0 1 2	0 0 9	0 0 9
0 3 7	9 8 2	0 1 2	0 1 2
3 9 5	1 1 2	1 1 2	0 3 7
0 1 2	7 3 3	4 2 8	0 5 5
0 0 9	1 6 4	7 3 3	1 1 2
9 8 2	3 9 5	0 3 7	1 6 4
7 3 3	0 5 5	0 5 5	3 7 7
1 1 2	0 3 7	1 6 4	3 9 5
4 2 8	3 7 7	3 7 7	4 2 8
3 7 7	4 2 8	9 8 2	7 3 3
0 5 5	0 0 9	3 9 5	9 8 2

Here we observe that elements are moved to their respective buckets **without any comparisons** made between any two elements, providing data parallelism. The cell processor's SPEs have little support for techniques like speculative execution and branch prediction, which makes **radix sort a better option** for the cell which involves no comparisons.

Radix sort is performed on the binary value of the elements, for example in a 32 bit unsigned integer, each pass could sorts the data with 8 bits at a time, so there are 4 passes each with 256 buckets. The following steps are involved in radix sort.

1. Count the number of elements in each of the 256 buckets for all the 4 passes.
2. Based on the bucket count of the current pass assign starting addresses for each bucket for the current pass.
3. Move the elements to their respective buckets.
4. Perform step 2 and 3 for other bytes too, ending with the most significant byte.

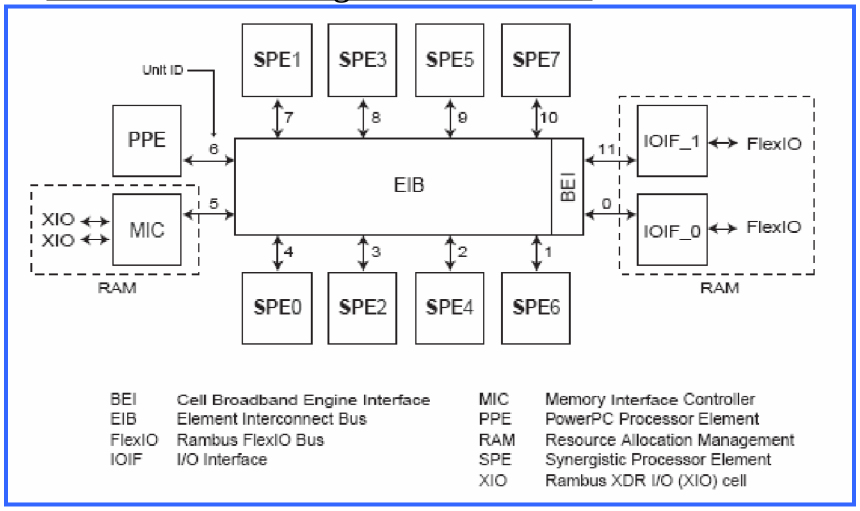
b. Parallel radix sort

Let N be the number of elements and P the number of processors, the following are the steps performed,

1. To each processor allocate N/P elements.
2. Each processor now calculates the bucket counts for the buckets.
3. Using this information from every processor, the starting addresses for each bucket of every processor is calculated by the controlling processor.
4. Each processor now moves their data accordingly.
5. Perform steps 1 through 4 for all the four bytes starting from least significant byte.

Here, unlike the serial algorithm we may not be able to calculate the bucket counts for all the four passes in the beginning itself, thus each processor brings in its data once for counting, once for moving, and writes back the data once, i.e. data is moved three times between main memory and the local memory, making it 12 times in all. Our algorithm aims at eliminating three of these passes, thus reducing the time complexity of the algorithm.

c. Cell Broadband Engine Architecture


SPE1
SPE3
SPE5
SPE7
Unit ID
PPE
6
7
8
9
10
11
EIB
BEI
5
0
XIO
XIO
MIC
IOIF_1
FlexIO
IOIF_0
FlexIO
4
3
2
1
RAM
RAM
SPE0
SPE2
SPE4
SPE6
BEI Cell Broadband Engine Interface
EIB Element Interconnect Bus
FlexIO Rambus FlexIO Bus
IOIF I/O Interface
MIC Memory Interface Controller
PPE PowerPC Processor Element
RAM Resource Allocation Management
SPE Synergistic Processor Element
XIO Rambus XDR I/O (XIO) cell

The PPE accesses the main storage (the effective-address space) with load and store instructions, the SPEs, in contrast, access main storage with Direct Memory Access (DMA) commands that move data and instructions between main storage and a private local memory, called a local store (LS). The eight identical SPEs are Single-Instruction, Multiple-Data (SIMD) processor elements that are optimized for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256-KB software-controlled local store (LS) for instructions and data, and a 128-bit, 128- entry unified register file.

2. The proposed algorithm

In our algorithm instead of performing the counting after the elements have been moved back to memory at the end of a pass, we calculate before sending the data, i.e. each processor maintains an array *next_pass*[*where*][*what*] where *where* is , i.e. it maintains the information of what data of the next pass is sent to where in the memory of this pass, we now know what data is where, which is all we need to calculate the starting addresses for each bucket of each processor for the next pass. The following are the steps performed

1. Allocate each processor N/P elements.
2. Each processor brings in its data and performs counting.
3. Destination addresses for the first pass is calculated.
4. Data is brought in, and before sending we calculate the radix for the next byte, i.e. let *index* be the radix for the current pass and *next_index* be the radix for the next pass, we increment the value *next_pass*[*index*][*next_index*].
5. At the end of this pass data is sorted based on least significant byte.
6. Using each processor's *next_pass* array we calculate starting addresses for the next pass, as we now know what elements are present where in the memory.
7. With these addresses the processors start off moving data.

In all the data is moved to processors once for counting, and for each pass brings in data once and wrights back the data one, making it 1+4+4=8 as opposed to 12 in the usual parallel radix sort.

3. Implementation Details

We have chosen to sort 16 bit *unsigned short int* with 4 bits at a time. It can be shown that it has the same computation and communication complexities as that of sorting 32 bit *unsigned int* with 8 bits as radix. The problem with sorting 32 bit Unsigned int is that the next_pass array needs to be of dimension 256x256 which is too big to fit in the Local store. A work around for this issue is part of future work.

4. Results

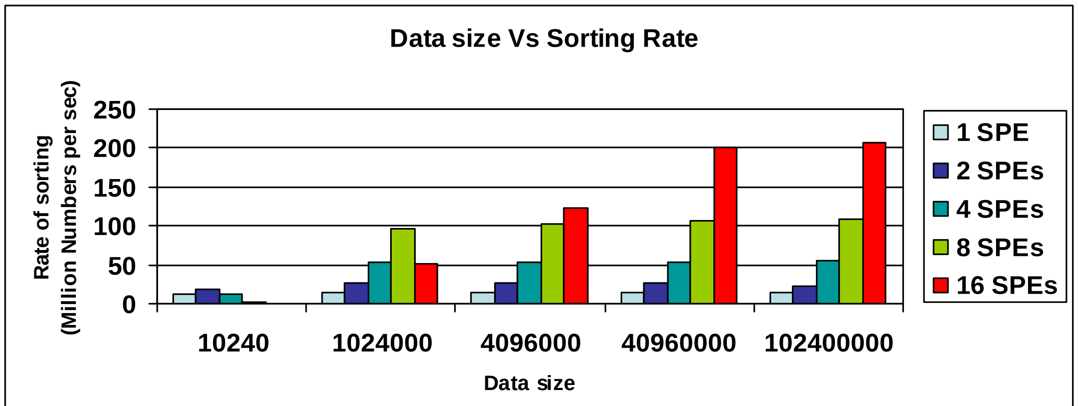


Figure 1

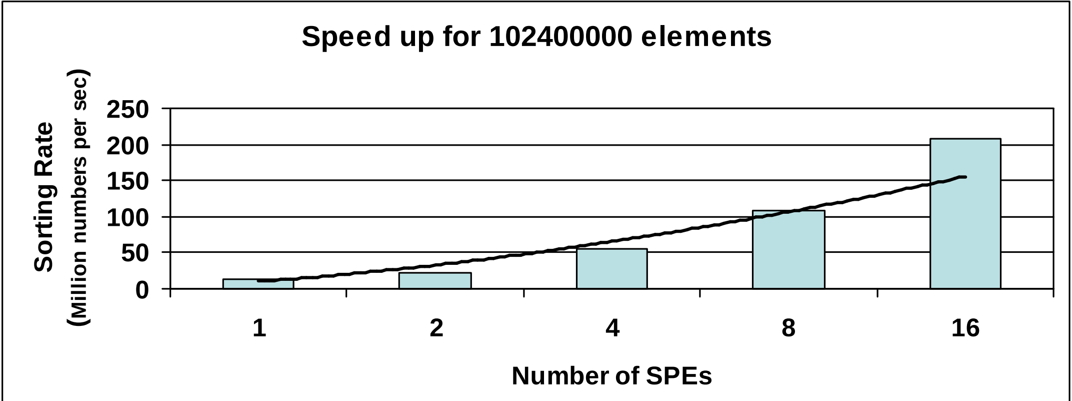


Figure 2

	10240	1024000	4096000	40960000	102400000
1 SPE	0.000872817	0.0738378	0.301164	2.93204	7.53521
2 SPEs	0.000566699	0.037794	0.150792	1.50938	3.77055
4 SPEs	0.000830772	0.0192813	0.075742	0.0755973	1.86562
8 SPEs	0.00450328	0.0107314	0.0403612	0.3814319	0.950866
16 SPEs	0.0160437	0.0201734	0.0333049	0.203431	0.493363

Time taken in seconds

Figure 1 shows the effect of parallelization for large data size. Figure 2 shows that the speed has nearly reached theoretical speedup. The table shows the time taken in seconds to sort elements and the number of SPEs used.

5. future work

- **a.** To move the computations performed by the PPE completely to SPEs.
- **b.** To calculate the radix we need to perform a shift and a logical AND, this operation can be vectorized.
- **c.** Currently we have not used multiple buffering. We have used blocking get and put commands. The non blocking MFC commands have the following requirements.
  1. DMA size: needs to be a multiple of 16 bytes.
  2. DMA source and destination addresses: needs 16 bytes alignment.

Here we propose a way to meet these requirements,
First the two main buffers in the MM are aligned using the compiler's align attribute, then whenever an SPEs need to send a bucket say *b* it appends *b.count* % 4 bogus data to the bucket. This helps us meet requirement one, this also makes the next bucket's starting address to be aligned by 16 bytes. We ensure in the next pass that this bogus data is processed by performing the same calculation again.

These three have been implemented in the single SPE code, in future we shall extend it to multiple processors.

6. Acknowledgements

We dedicate this work to **Bhagwan Sri Sathya Sai Baba**, the Chancellor of the Sri Sathya Sai University. We also would like to acknowledge the guidance and support given by Dr. Ashok Srinivasan, Department of Computer Science, Florida State University. We 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.

7. References

1. Cell broadband engine architecture document.
2. "Load Balanced Parallel Radix Sort" by Andrew Sohn and Yuetsu Kodama.