Word-size Parallelism*

Andrej Brodnik
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

e-mail: `ABrodnik@WatDragon.UWaterloo.CA`

*This work was supported in part by ITRC project "Algorithms and Data Structures".

1 Background

The notion of *word-size parallelism* came up in a context of some other work on geometrical problems on an integer grid. In that work we had to handle many pieces of data which were relatively small in comparison to the size of the word. Because we were able to align these pieces of data in the same word and perform on all of them operations simultaneously, we achieved a substantial speed up in comparison to the sequential algorithm.

In fact, this kind of parallelism can be found in usual processor instructions sets. For example, it is perfectly justifiable to consider any bitwise Boolean operation as parallel operations performed on individual bits of the word. In this paper we generalize such a parallelism by using larger blocks of bits. The parallelism has, with latest 32 and 64 bit generation of processors, even greater practical significance (cf. [7]).

In order, to make our results more general we denote the word size by m. Next, we use the notation x.b[i] to represent the i^{th} bit of the word x, where $0 \leq i < m$. The bits are further grouped into blocks numbered from 1 to l, on which we apply operations in parallel. Finally, these operations represent a single instruction from the processor's point of view and we assume that the processor performs any operation from its instruction set in one unit of time.

It is reasonable to add some theoretical background about the model of computation we are dealing with. The model we use is a *cell probe model* as it was defined in [2, 3, 8] with a word size m and a usual instruction set including multiplication.

2 New technique

Word-size parallelism is a programming technique, which is based on a splitting of the word in a number of blocks and on a parallel performing of non-interfering operations on these blocks. Obviously, more blocks we have better speed-up we achieve in comparison to a sequential algorithm.

The generic form of function is presented in Figure 1. It deals with an argument x split into l blocks and

```
GENERIC PROCEDURE Technique (x);
  FOR i:=1 TO l DO PARALLEL
    answer[i]:= Apply (F_i, x[i]);
  END;
  y:= C (answer);
  RETURN y;
END Technique;
```

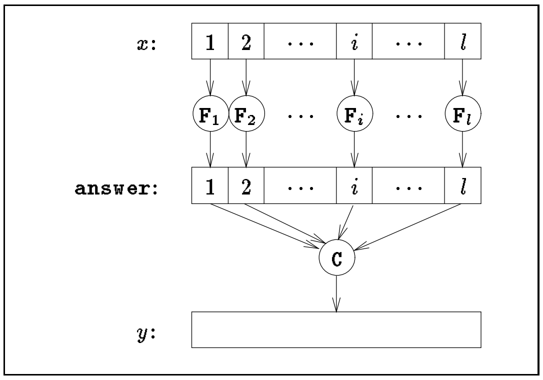

Figure 1: Word-size parallel computation: computation of a number of operations in "parallel" and combination of their results at the end.

applies on individual blocks (possibly different) operations $\mathtt{F}_i$. Partial results of individual operations are at the end combined together using function $\mathtt{C}(\cdot)$ to get the final result of the function.

In the next section we present a few different instantiations of the generic function which differ in the implementation of a parallel `FOR` loop with its functions $\mathtt{F}_i$ and in the implementation of a final function $\mathtt{C}(\cdot)$.

3 Example

The problem we use as an example is reversing the bits in a word. It requires to implement a function `Reverse` which reverses the bits in the argument x into a result y in such a way that $y.\mathtt{b}[i] = x.\mathtt{b}[m-1-i]$ for $0 \leq i < m$ (see Figure 2). Note, that from a theoretical point of view there is a simple $\Omega(m^2)$ lower bound for one-head one-tape Turing machines which can be derived from the lower bound on a palindrome ([5]), but it does not directly apply to our model.

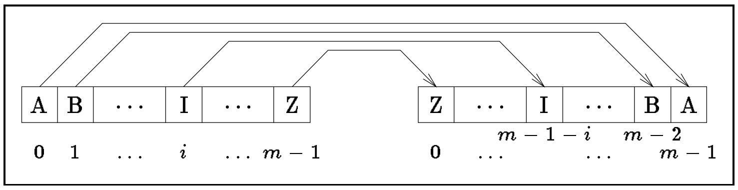

Figure 2: Reversing of the word.

A version of the problem was already studied in a context of FFT (cf. [1, 6]) but there the problem was to reverse n words from some array.

3.1 Solutions

For the sake of completeness we first mention a straightforward algorithm, which in $\lfloor m/2 \rfloor$ steps swaps corresponding bits in a word and, hence, performs about $4 \cdot \lfloor m/2 \rfloor$ operations in total. On the contrary to the remaining two algorithms, this one does not use word-size parallelism.

The second algorithm we describe uses the word-size parallelism as a basic operation iterated $\lceil \lg m \rceil$ times. The basic operation, in detail, at j^{th} iteration takes x split into blocks of size $2k = 2^j$ bits and in each block swaps lower k bits with the upper k bits. It is not hard to see that this algorithm really reverses the word (see Figure 3 for reversing of an 8 bit word). Going back to the generic function in Figure 1 we see,

```
PROCEDURE LogReverse (x);
  FOR i:=1 TO ⌈lg m⌉ DO
    tmp:= x AND mask[i];
    x:= x AND Negate (mask[i]);
    x:= tmp OR ShiftLeft (tmp, i);
  END;
  RETURN x;
END LogReverse;
```

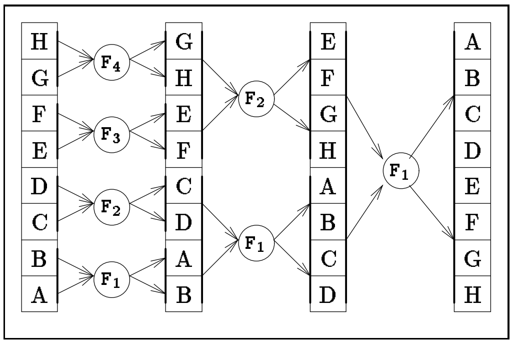

Figure 3: The logarithmic solution.

that in the basic operation the final function `C` is omitted, while all functions $\mathtt{F}_i$ are the same – they swap lower k bits with upper k bits in respective blocks. More precisely, the parallel `FOR` loop is in the algorithm implemented by three operations inside the sequential `FOR` loop of the algorithm in Figure 3. Therefore, the complete algorithm iterates $\lceil \lg m \rceil$ times the basic operation and therefore performs in total approximately $6 \cdot \lceil \lg m \rceil$ operations.

The last solution we discuss performs a constant number of operations regardless of the size of the word, provided that the upper $m - s$ bits, where $s < \lfloor \sqrt{m} \rfloor - 1$, are 0. The algorithm works in two steps. In the first step it distributes s least significant bits across the word and applies proper mask to get these s bits in equidistant positions, but in the reverse order. This step is in fact an implementation of the parallel `FOR` loop of the generic function from Figure 1. The second step, the implementation of a function $\mathtt{C}(\cdot)$, gathers bits back together. It uses the result by Fredman and Willard [4] which guarantees that if y is a word with set bits on equidistant positions only, then for some constants a and b the expression `ShiftRight` $(a \cdot yANDb$, `m`) gathers these bits together on least significant positions and in the same order as they were in y. In addition, we also assume that we can perform a double precision arithmetic. The algorithm is shown in Figure 4 and uses additional constants $d = \sum_{i=0}^{s-1} 2^{i(s+1)}$ and `mask`$= \sum_{i=0}^{s-1} 2^{i(s+1)+s-1-i}$. They are combinations of constants used in parallel functions. So, function $\mathtt{F}_i$ shifts s least significant bits for $i(s+1)$ positions to the left and then masks out all but $(s-1-i)^{\text{th}}$ bit. In total the function 5 operations for any value of m.

```
PROCEDURE ConstantReverse (x);
  y:= (x * d) AND mask;
  y:= ShiftRight (a * y AND b, m);
  RETURN y;
END ConstantReverse;
```

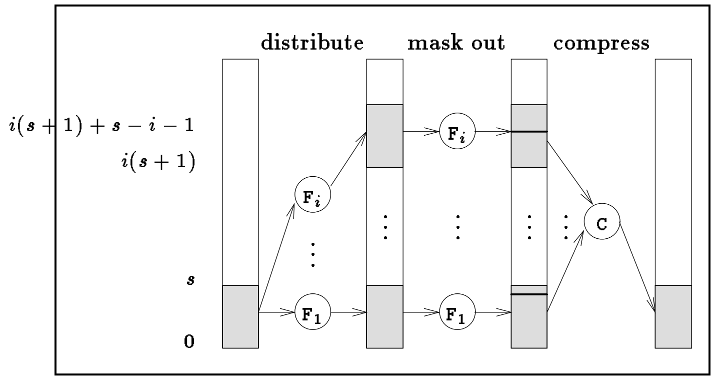

Figure 4: The constant solution.

4 Conclusions

We introduced a programming technique called *word-size parallelism* and show its usage on two algorithms which reverse a word. One of the algorithms used approximately $6\lceil \lg m \rceil$ operations while the other only 5 operations provided that the argument has only set bits in the first $\lfloor\sqrt{m}\rfloor - 1$ positions. Furthermore, since none of operations used by the second algorithm is branching, it makes the algorithm even more feasible for modern pipeline architectures.

From the theoretical point of view the lower bound for reversing of the word on a cell probe model with a multiplication remains an open problem. Obviously, it depends on the amount of work done in precomputation and on the number of bits in the word we want to reverse. However, from the result presented we also see the bound remains constant even for a non-constant number of reversed bits.

Acknowledgment

I would like to thank my supervisor Ian Munro for his help in shaping up this work.

References

[1] A.C. Elster. Fast bit-reversal algorithms. In *1989 International Conference on Acoustics, Speech, and Signal Processing*, pages 1099–1102, Glasgow, Scotland, 1989.

[2] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 1 – 66. Elsevier, Amsterdam, Holland, 1990.

[3] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 1989.

[4] M.L. Fredman and D.E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *22nd ACM Symposium on Theory of Computing*, pages 1–7, Baltimore, Maryland, 1990.

[5] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[6] J. Jeong and W.J. Williams. A fast recursive bit-reversal algorithm. In *1990 International Conference on Acoustics, Speech, and Signal Processing*, pages 1511–1514, Albuquerque, New Mexico, 1990.

[7] Geppert L. Not your father's CPU. *IEEE Spectrum*, 30(12):20–23, December 1993.

[8] A.C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.