Parallelism through Digital Circuit Design

John T. O'Donnell[1]

University of Glasgow, Department of Computing Science
Glasgow G12 8QQ, United Kingdom
jtod@dcs.gla.ac.uk

Abstract. Two ways to exploit chips with a very large number of transistors are multicore processors and programmable logic chips. Some data parallel algorithms can be executed efficiently on ordinary parallel computers, including multicores. A class of data parallel algorithms is identified which have characteristics that make implementation on multiprocessors inefficient, but they are well suited for direct design as digital circuits. This leads to a programming model called *circuit parallelism*. The characteristics of circuit parallel algorithms are discussed, and a prototype system for supporting them is described.

Keywords. circuit parallelism, data parallelism, FPGA

1 Introduction

We commonly assume a sharp distinction between computer hardware and software: the digital circuit designer takes logic gates as primitives to build processors, while the programmer takes processors as primitives to build applications. Specialists in one of these domains seldom cross into the other.

For many years this division of labour has sufficed. The increased circuit density that resulted from improved chip manufacturing has enabled circuit designers to produce ever faster processors, and everyone benefited. That era is ending, and the current trend is toward multiple processors per chip rather than bigger processors.

But there is an alternative way to exploit all the extra transistors now available: you can use them to provide special assistance to ordinary processors, or even to design circuits that solve problems directly. Thus the programmer partitions an algorithm into a part that will run on a CPU, and a part that will execute as a digital circuit.

A prototype system along these lines is discussed in this paper. The central components are the use of FPGA technology to reduce the cost of circuit design, the use of a functional language, Haskell, to program the software and specify the hardware, and a set of interfacing tools to connect both parts together. At the present time there is no automatic support for partitioning the algorithm into software and hardware, but it is possible to write it entirely in Haskell, refactor the program, and reason about correctness.

This paper presents an overall picture that still lies partly in the future, and discusses progress that has been made already. The remaining sections discuss the effect that improvements in chip fabrication is having on computer architecture. The data parallel programming model is discussed, and some of its common characteristics are identified. The paper then describes a class of algorithm which is, strictly speaking, data parallel yet which cannot be executed efficiently on most architectures intended for data parallel applications. The term *circuit parallelism* is proposed as a name for this programming model. A prototype system with an FPGA and support software has been implemented and demonstrated using a simple application of circuit parallelism.

2 The changing context

As fabrication technologies for integrated circuits have improved over the past forty years, the number of transistors per chip has increased exponentially. For much of that time, this has been accompanied by a steady increase in processor speed. The density of chips continues to grow, but processor speeds are stabilizing. These two trends are changing the context in which we design and use computer systems.

The selling point for new personal computers has long been faster clock speeds. These were achieved through the use of faster components, reductions in critical path depth, and the use of more components to achieve higher speed. But clock speed is not improving as quickly as it has in last few decades.

Furthermore, the main techniques for using extra transistors to reduce the number of clocks needed to run a program (such as pipelining, cache, superscalar, etc.) have already been applied, and cannot be applied again for further improvement. Again, however, most of these techniques are already applied routinely, and they cannot be applied again for further improvement.

The conclusion is that individual processors will continue to get faster, but the rate of improvement is diminishing.

On the other hand, the number of transistors that can be placed on a chip is far larger than needed to form a processor — even a large one. The challenge now in computer architecture is in finding useful ways to exploit such large numbers of transistors.

The most obvious way to proceed is simply to place several processors on one chip. This is called *multicore* architecture, and the number of processor cores per chip is predicted to increase steadily. Since all PCs will soon have multicores available, a challenge in parallel programming is to find effective ways to use them.

This paper argues that an alternative to multicores is more effective for a certain class of algorithm. The following sections provide some background on the programming model of data parallelism, and then introduce a special case of data parallelism which we call *circuit parallelism*.

3 Data parallelism

Several early high performance computers supported parallelism organised around data structures. This was a natural development, as many scientific applications spend most of their time executing nested loops over arrays.

The Control Data Star 100 computer provides vector instructions that operate on arrays in memory. The processor and memory systems were designed to support efficient pipelining, so that an inner loop operation (e.g. computing `tmp[i] := x[i]+y[i]`) would execute faster than using a conventional instruction set.

A further refinement appeared in the Cray 1 [1], which has a set of eight vector registers, called V registers. Each V register contains 64 elements, where each element can hold a 64 bit floating point number. The V registers allow complex inner loop calculations to be performed significantly faster, as the vector elements need to be fetched from memory only once, but can be used as often as needed to evaluate expressions. The V registers constitute a tradeoff: they require vector operations to be broken into chunks of 64 elements, but they reduce memory traffic and increase parallelism in the evaluation of complex arithmetic expressions.

Vector supercomputers are effective for programs whose execution time is dominated by a specific form of inner loop, but some programs have data parallelism with a slightly different structure, making them poorly suited for vector machines. This observation led to the development of general SIMD architectures.

The Massively Parallel Processor [2] consists of a square array of small processors (also called processing elements or PEs). The word size of the processors is minimal: their registers and internal data paths are 1 bit wide. The PEs lack the ability to fetch and decode instructions; instead, the program runs in a control processor (which is a conventional computer). The control processor can issue instructions, in the form of control signals, to the array of PEs. Using the terminology of processor organisation, each PE is a single 1-bit datapath, and the control unit is emulated by the main computer.

Since all the PEs receive the same control signals on every clock cycle, they are limited to executing the same operation—hence the “single instruction, multiple data” architecture. However, the architecture does provide for conditional execution. Each PE contains a 1-bit mask register. Some of the instructions are conditional: these are executed in PEs where the mask is 1 but ignored on the others. Thus a loop containing an if statement can be executed in parallel: first the mask registers are set to the condition; the then-clause is executed conditionally; the mask register is inverted; and finally the else-clause is executed conditionally.

The PEs are connected by two interconnection networks. Each PE has a direct connection to its north, east, south and west neighbors, from which it can read a bit. In addition, there is a tree of logical or-gates whose output can be read by the control processor. This can be used in a variety of ways; in particular it enables the control processor to select a PE and then read a bit from it.

The MPP design exhibits an interesting tradeoff. The extremely small PEs are quite slow, but their simplicity enables the system to have a large number of PEs. Using early 1980s technology, it was possible to fabricate eight PEs per chip, and the MPP contained 16,384 PEs at a time when a parallel computer with twenty processors was considered very large.

Another interesting tradeoff concerns the match between the architecture and the application. Bit-serial machines like the MPP are slow at floating point computation, but some algorithms, use arrays of small integers. The MPP was reported to give much better performance than the Cray 1 on a radar signal processing application, where the data consisted of small pixels and the floating point hardware was irrelevant.

The Thinking Machines Connection Machines CM-1, CM-2, and CM-200 were similar to the MPP. They offered a larger number of PEs and a richer interconnection network, but the basic organisation—bit serial PEs with a SIMD architecture—was the same.

All the architectures discussed in this section were intended to speed up the inner loops that dominate execution time in many scientific applications. To achieve this, the architectures contain functional units or processing elements that operate simultaneously on many data elements, while the software remains sequential. Thus the parallelism is organised around the aggregate data structures.

In the original papers on these early parallel systems, they architectures were often called "vector" or "SIMD". Eventually, as the architectures became more general, the term "data parallel" became more popular.

4 Data parallelism on multiprocessors

A practical difficulty with SIMD architectures was that conventional microprocessors were improving every year, while it was too costly to keep redesigning the special purpose PEs needed for SIMD. A multiprocessor, built using standard commercial microprocessors, can offer a better price-performance ratio in an environment where the speed of microprocessors is increasing rapidly. However, this argument carried more weight in the 1980s and 1990s than it does now, when the rate of improvement in clock speeds is declining.

Thinking Machines Corp. abandoned the SIMD approach (the CM-200 and its predecessors) in favor of a coarse grain multiprocessor, with a relatively small number of powerful microprocessors. The CM-5 has an array of powerful SPARC processors connected with a fat-tree network. Thinking Machines claimed [3] that the CM-5 was optimised for efficient data parallel execution, but could also support SPMD and task parallelism.

5 Characteristics of data parallelism

"Data parallelism" is a rather vague term; it lacks a single, universally accepted and immutable definition. Original papers on early architectures seem to equate data parallelism with a SIMD organisation, while more recent papers treat data parallelism as a special case of SPMD, or just a way of thinking about an algorithm that will be executed on an MIMD architecture.

However, many data parallel programs share three characteristics that often go unmentioned, but which are significant in choosing a suitable architecture.

The first characteristic is a restriction to simple dense arrays and to basic aggregate operations. Many data parallel algorithms are simple iterations over dense arrays (supported by vector computers). Many of the exceptions to this are slightly more general iterations over dense arrays (supported by SIMD architectures).

The second characteristic is a restriction to simple aggregate operations. For example, data parallel systems commonly support folds and scans, but they restrict the function being folded or scanned to a small set (such as logical and, logical or, integer addition, integer multiplication, floating addition, floating multiplication, double addition, and double multiplication).

The third characteristic of most widely-known data parallel algorithms is that they use the data structures to organise the parallelism, but the parallelisation does not introduce any extra work. That is, the actual set of computations to be performed is the same regardless of whether the parallelism is arranged according to data or tasks.

However, there do exist data parallel algorithms with quite different characteristics. For example, it is possible to apply data parallelism to highly irregular data structures, including nested lists and trees. A key technique for doing this is the provision of general aggregate operations. APSA (applicative programming system architecture) [4] [5] [6] is a SIMD design with a programmable tree interconnection network that supports general list processing. It has a general tree sweep operation; this can be used to implement parallel folds and scans with *general user-defined* associative functions. APSA supported a form of algorithm that more recently came to be called "nested data parallelism".

6 Circuit parallelism

Several data parallel algorithms have appeared which violate all three of the common characteristics identified in the previous section. In addition to using general parallel operations over irregular nested data structures, they also introduce more computations than would be performed by a sequential or task-parallel algorithm—yet the extra work yields a net speedup on a suitable architecture.

One such algorithm is the ESF array [7], a form of pure functional array that may be sparse, and which may be extended dynamically. The implementation uses associative searching to locate array elements. Every value is associated with tagging information that enables it to be found in unit time by global matching. A value, along with its tags, is stored in a processing element that has the ability to perform tag comparisons in parallel with all the other PEs. A simple fine grain architecture can execute this algorithm efficiently. The processing elements require only a around a thousand logic gates; more power per PE would not help, but a large number of PEs are required.

Several more algorithms with similar characteristics have been worked out. These include an embedding of restricted tasks within a data parallel setting [8] and a sorting algorithm using memoised selection.

The following code fragment, taken from the sorter using memoised selection, gives a concrete idea of what is involved in this class of algorithm. The full algorithm is not explained here (see [9]). The split operation is similar to splitting in quicksort: a splitting value `s` is used to divide the elements of a set into a subset of smaller and a subset of larger elements. The important point to note about this split operation is that *it contains no loops*: it is straight-line code. The operation `n <- count sr` uses the tree interconnection network to count the number of PEs that satisfy a certain condition; the calculation `let k = l + n` is performed in the control processor; all the other lines of code are microinstructions that are performed locally within each PE, in parallel. The entire split operation requires a constant 15 clock cycles, regardless of the size of the data structure being split.

```
split :: IORef State -> Value -> Bound -> Bound -> IO ()
split sr s l u =
  do setSelect sr True
     match sr lb l
     match sr ub u
     save sr                -- iff cell matchs (lb,ub) bounds
     compareVal sr (<) s    -- select indicates match with val<splitter
     n <- count sr          -- number of matches < splitter
     let k = l + n
     condSetUB sr (k-1)     -- where val<s, update the upper bound
     restore sr
     compareVal sr (>) s    -- where val>s,
     condSetLB sr (k+1)     --   update the lower bound
     restore sr
     compareVal sr (==) s   -- where val=s,
     condSetLB sr k         --   update the lower bound
     condSetUB sr k         --   and the upper bond
```

The crucial point about these algorithms is that every operation involves a small calculation on *every data element in the memory*, not a large calculation on just a few items. A fine grain SIMD architecture like the CM-200, or APSA, can execute the algorithm efficiently, but a coarse grain one like the CM-5 cannot. And a digital circuit, tuned specifically for the application, would be far more efficient even than the fine grain SIMD machines.

Conventional data parallel algorithms perform the same calculations that would be performed by a sequential program. Algorithms like ESF arrays are completely different in their computational requirements. They are indeed data parallel, but this classification is misleading.

This argument motivates the definition of a new model of parallelism: *circuit parallelism* is an organisation where parallel computations are performed in separate digital circuits, which may be extremely fine grain. It is perfectly acceptable to introduce extra computations that would not be needed by a sequential program, as long as the overhead is minimised by the inherent parallelism of the circuit.

Consider an application based on associative searching (e.g. the ESF array). Such algorithms generally require parallel scans (or, even better, parallel tree sweeps) using complicated auxiliary functions (*not* just folding the addition or multiplication operators). The parallel sweep or scan can be implemented in logarithmic time using a tree-structured circuit. This seems to be a fairly slow operation compared with standard microcomputer hardware, but actually it is not. A computer's memory requires an address decoder, which is essentially a tree circuit that performs a trivial computation in each node (go left or right, depending on the address bit). The circuit-parallel algorithms also require a tree-structured circuit; the difference is only that a more complex circuit is needed in the tree nodes, but this introduces only a small constant factor slowdown.

Another way of looking at it is that a conventional RAM memory is a circuit-parallel algorithm, realised in hardware, with trivial processing elements (the memory bits) and a trivial algorithm in each tree node (just a demultiplexor). More powerful circuit-parallel algorithms replace these trivial circuits with more general ones.

Circuit-parallel algorithms would be hopelessly inefficient on a coarse grain multiprocessor, even with very fast processors. These algorithms fit reasonably well on general SIMD architectures, but they are far more effective if they can be implemented directly as digital circuits, benefiting from the extraordinarily high degree of parallelism inherent in circuits.

7 A prototype FPGA system architecture

It is usually too expensive to design a digital circuit and fabricate it in hardware just to speed up a particular computer program. Programmable logic devices, however, provide the ability to emulate a digital circuit while retaining nearly all the parallelism.

If a program is enhanced using circuit parallelism, the resulting system will contain a conventional processor (or multiprocessor) connected to an FPGA which is running the parallel circuit. A number of technical issues must be solved to make this feasible: a compatible programming style for the software and the circuit parts of the algorithm is needed, and an interface between the processor and the FPGA is required to enable them to communicate with each other. Furthermore, this interface consists partly of software (for the computer) and hardware (to be emulated in the FPGA).

A prototype system along these lines has been implemented [10] using a combination of Haskell (for the software), Hydra (for the circuit), an Altera FPGA (for emulating the circuit), and a serial cable with software and hardware drivers (for the communication).

The system consists of a CPU connected to an FPGA (Figure 7). In the prototype implementation, a slow serial cable connects the chips. The cable is connected to a hardware UART in the computer, and to a programmed UART circuit running in the FPGA.

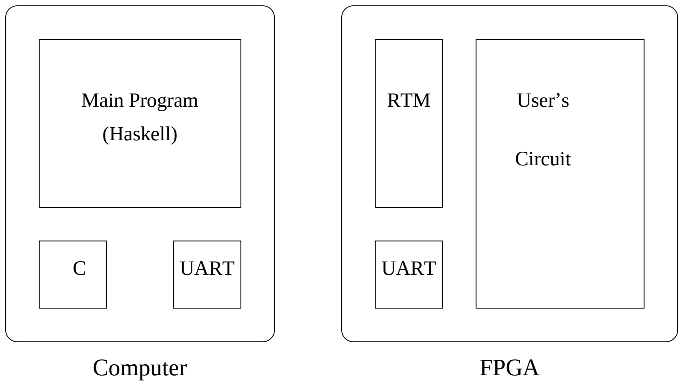

Fig. 1. Prototype system architecture.

The computer runs the main program; in our experiments this is written in Haskell, but any ordinary programming language can be used. For convenience, the input/output is coordinated by a small C program that communicates with the main program using the Haskell foreign function interface, and with the UART using the operating system's application programming interface (API).

The digital circuit is specified in Hydra [11], a hardware description language that is embedded within Haskell. Hydra allows the simultaneous specification of the structure and behaviour of a digital circuit, and the notation is compatible with functional programs written in a combinator style.

The application program is partitioned into two parts, both written in Haskell (as Hydra is indeed just a specialised form of Haskell). The software portion is compiled by ghc to produce a machine language program for the computer, while the hardware portion is processed by Hydra in order to produce an executable file for the FPGA. This conversion is performed in two steps: Hydra generates a VHDL definition of the circuit, and the chip vendor's software tools convert the VHDL into the necessary binary.

8 Conclusion

Recent trends in fabrication of VLSI chips have made enormous numbers of transistors available—far too many to use effectively in individual processors. Therefore the manufacturers of microprocessors are advocating multicore architectures.

Multicores are useful for task parallelism, and perhaps for conventional data parallelism, but they are not the only way to make use of more transistors. An alternative is FPGAs, or other programmable logic devices, which benefit automatically from increasing numbers of transistors, and which make all the parallelism of digital circuits available to the programmer.

There is a special form of data parallelism with different characteristics from ordinary data parallel algorithms. These algorithms require large numbers of fine grain processors, with programmable interconnection networks. They may require more primitive computations to be performed than sequential algorithms, but if all the computations can be performed in parallel there is a net speedup. This paper proposes the term *circuit parallelism* for this variant of data parallelism. Although ordinary data parallel programs can run efficiently on multiprocessors, algorithms with circuit parallelism require the extraordinary degree of parallelism that is available only with digital circuits. They can execute efficiently on FPGAs, but not on multicores.

Circuit design is not just for computer engineers! It isn't difficult, with the use of suitable hardware description languages, and it can give outstanding performance.

References

1. Cray Research Inc.: Cray 1 Computer System Hardware Reference Manual. (1977)
2. Potter, J.L.: The Massively Parallel Processor. The MIT Press (1985)
3. Inc., C.R.: CM-5 User's Guide. (1993)
4. O'Donnell, J.: A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management. PhD thesis, University of Iowa, Iowa City (1981) Technical Report 81-5.
5. O'Donnell, J.: Parallel VLSI architecture emulation and the organization of APSA/MPP. In: Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation. Volume NASA Conference Publication 2478., Code NTT-4, Washington, D.C. 20546-0001, Science and Technical Information Office, National Aeronautics and Space Administration (1986) 75–84
6. O'Donnell, J., Bridges, T., Kitchel, S.: A VLSI implementation of an architecture for applicative programming. In: Proceedings of the Conference on Frontiers in Computing, Elsevier (1987) 315–330
7. O'Donnell, J.: Data parallel implementation of Extensible Sparse Functional Arrays. In: Parallel Architectures and Languages Europe. Volume 694 of LNCS., Springer-Verlag (1993) 68–79
8. O'Donnell, J.T.: Supporting tasks with adaptive groups in data parallel programming. International Journal of Computational Science and ENgineering (IJCSE) (2005)
9. O'Donnell, J.: Functional microprogramming for a data parallel architecture. In: Proceedings of the 1988 Glasgow Workshop on Functional Programming, Computing Science Department, University of Glasgow (1988) 124–145
10. Koltes, A., O'Donnell, J.: Circuit parallelism in haskell programs. In: IFL 2007 Draft Proceedings, University of Freiburg (2007) extended abstract.
11. O'Donnell, J.: Overview of Hydra: A concurrent language for synchronous digital circuit design. International Journal of Information **9** (2006) 249–264