Introduction

Parallel Processing has had a major impact on the development of computer science. There is extensive literature on parallel architectures, parallel programming languages and parallel algorithms which has achieved insight into the capabilities and limitations of parallel machines.

However, the diversity and complexity of parallel architectures have created a fundamental challenge for programmers, viz. the efficient mapping on algorithms to the parallel environment. Although efficient practical algorithms for problems on particular parallel architectures are already known, still, the active area of current research is the development of a general methodology for the design of parallel algorithms that are practical and efficient on a wide variety of architectures.

This thesis is a further step in that direction. We propose a paradigm for structuring and designing programs for parallel computing. This paradigm is described informally, and advocated by means of a collection of case studies on parallel architecture which are massive. The inspirational motivation for our paradigm is the drive to initiate a uniform problem-solving method for eliciting the underlying algebraic structure. The existence of some algebraic structure is particularly important in the context of parallel computing, insofar as parallel computers typically perform more efficiently on highly structured computations than they do on unstructured computations.

Typically, the algebraic structure takes the form of a group of transformations that ignore many invariant salient computational characteristics which help in defining the basic problem. In addition to using parallel algorithms for exploiting this symmetry, an underlying theme permeates this thesis: the primacy of symmetry considerations in a broad range of applications, including string matching, particle simulation, and communication primitives.

Our method comprises of the following two main steps:

Firstly, the problem is translated into a generalized matrix multiplication problem. In doing so, addition and multiplication are replaced by more generalised operations, using Extended Karnaugh-Map representation, Matrix Vector Multiplication and comparative Study of different algorithms. This carries several advantages like the probable use of highly optimized

linear algebra libraries for parallel machines, and new formulation of multi-linear algebraic techniques for considerable amount of hardware.

Secondly, the problem has been translated into matrix symmetry, and this will help in capturing the original problem. Mathematically, the associated matrix will commute with a group of permutation matrices. These invariant matrices admit several parallelizable techniques for their efficient multiplication.

The first technique is simply factorization; i.e. the matrix is factorised into smaller matrices that represent primitive operations supported by the target architecture. This technique emphasizes the use of the multidimensional matrix-matrix product to extract parallelism. This is strongly influenced by the success of the formulation of multidimensional matrix multiplication for parallel algorithms.

The second technique for the manipulation of invariant matrices is called orbit decomposition. Orbit decomposition is a formalization of the familiar technique of caching computations in order to reuse the data later. Orbit decomposition can sometimes induce a particular routing pattern in the parallel architecture: a Cayley graph. This Cayley graph is a graphical representation of the symmetry inherent in the original problem. As it happens, the network connecting the individual processors in a number of classes of parallel machines is also, in many cases, a Cayley graph; our methodology thereby gives rise to an interaction between a Cayley graph arising at the software level, from symmetry considerations, and a Cayley graph arising at the hardware level, from the interconnection network of the processors.

The third technique we use for manipulation of invariant matrices is group Fourier transforms. These generalize the familiar discrete Fourier transformations, the Cooley-Tukey implementation of which has been inertial in many areas of computer science. Group Fourier transforms are based on techniques of group representation theory, which can be loosely viewed as the use of matrices to model symmetry, and have undergone energetic development by a number of earlier researchers.

This earlier work has demonstrated many applications for general group Fourier transforms, in areas such as matrix multiplication, machine learning, VLSI design, vision, random walks, and graph algorithms, and has provided fast algorithms for many classes of finite groups. Although some amount of mathematical machinery, primarily basic representation theory, is necessary to understand these fast algorithms, we have tried to carefully encapsulate

areas of this thesis which require such knowledge. Their salient feature for our purposes is that they allow fast parallel algorithms for multiplication by invariant matrices. Work-efficient parallel group Fourier transform algorithms are introduced in this thesis, improving on several suboptimal constructions in the literature. These three techniques |factorization, orbit-decomposition, and group Fourier transforms| are the tools we use to exploit symmetry.

Before continuing with the overview of the thesis, we emphasize two main points: First, it should be clear that there are many classes of problems to which the paradigm we propose does not readily apply. For example, problems with unstructured data-access patterns, or data-access patterns that are not known at compile-time would be a poor match for this approach. Typical examples of such problems include open-ear decomposition in graph problems, forward alpha-beta search with pruning heuristics, and several classes of combinatorial optimization problems.

Nevertheless, our paradigm can be used to solve structured sub problems of an unstructured problem. For example, the adaptive multiple method is a dynamic tree algorithm for particle simulation to which we cannot usefully apply our ideas, but even the adaptive multiple method must, at some point, call a direct \brute-force" particle simulation routine at which point our ideas apply. Similarly, although the parallelization of alpha-beta chess programs is beyond the scope of this work, the leaf evaluation subroutines of such programs typically rely on a specialized endgame module of the type described in this thesis.

Second, we remark that the field of parallel processing is changing so fast, and with such a complex interconnection between economic, technological, hardware, and software factors, that we cannot claim that our paradigm is the last word on the topic. In Part II, therefore, we will not focus solely on our parallelization techniques. Instead, we will also describe a number of applications that, we hope, will be seen to have a beauty and interest independent of the ultimate viability of our main program; these case studies will also be useful considered only as examples of the successful design and implementation of parallel algorithms. Before continuing with the overview of the thesis, we emphasize brief introduction about multidimensional matrix operations and Extended Karnaugh-Map:

Multi-dimensional matrix multiplication and array operations are used in a large number of important scientific codes, including finite element methods, molecular dynamics, and climate modeling. Various methods have been proposed for the efficient implementation of these Multi-

dimensional matrix multiplication and array operations, but most of these methods tend to focus on the 2-dimensional arrays. “When extended to higher dimensional arrays, these methods usually do not perform well. Hence, designing efficient algorithms for multidimensional matrix operations becomes an important issue.” [133]

It is to be noted that the shared memory also functions as a communication medium for the processors. Here each step of an algorithm consists of the following phases:

▪ Read: Can read up to N processes simultaneously in parallel from N locations and store their value in local registers.

▪ Compute: N processors perform basic arithmetic or logical operations on the value in their registers.

▪ Write: N processors can write simultaneously into N memory locations from their registers. This PRAM model can be further sub-divided into four categories based on the way simultaneous memory accesses are handled:

▪ Exclusive Read, Exclusive Write (EREW) PRAM. In this model every access to a memory location (Read/Write) has to be exclusive.

▪ Concurrent Read, Exclusive Write (CREW) PRAM. In this model only write operations to memory location are exclusive whereas two or more processors can concurrently read from the memory locations are exclusive.

▪ Exclusive Read, Concurrent Write (ERCW) PRAM. This model allows multiple processors to concurrently write to the same location, whereas, the read operations are exclusive. Concurrent Read, Concurrent Write (CRCW) PRAM. This model allows both multiple read and multiple write operations to a memory location. It is the most powerful of the four models. During read operations all processors reading from a particular memory location read the same value, whereas, during write operation many processors try to write different values to the same memory location. So, this model has to specify precisely the value that is to

be written to the memory location. So, we specify some protocols which identify the value that is to be written to a memory location. They are:

▪ Priority CW: Here only the processor with highest priority can succeed in writing its value to the memory location.

▪Common CW: Here the processors are given a chance to write to a memory location if and only if they have the same value. o Arbitrary CW: Here if one processor succeeds in writing to memory location it is arbitrarily chosen without affecting the correctness of the algorithm.

▪ Combining CW: Here there is a function that maps the multiple value that the processors try to write a single value that is actually written into the memory location. Interconnection Networks – We know that in PRAM, all exchanges of data among processes take place through shared memory. There is also another way for the processors to communicate i.e. via direct links. In this method instead of shared memory the M locations of memory are distributed among N processors. So the local memory of each processor now contains M/N locations. Combinational Circuits – A combinational circuit can be viewed as a device that has a set of input lines on one end and set of output lines on the other. Such type of circuits are made of interconnected components arranged in columns called stages, each component has a fixed number of input lines called fan in and fixed number of output lines called fan out. After each component receives its input a simple arithmetical or logical operation is performed in one unit time and result is produced as output. A schematic diagram of a combinational circuit is shown below. For convenience of Fan In and Fan Out are assumed to be 2. An important feature of combinational circuit is that it has no feedback. The important parameters used to analyze a combinational circuit are: Size: Refers to the number of components used in the combinational circuit.

Depth: It is the number of stages in the combinational circuit i.e. the maximum number of components on a path from input to output.

Width: It is the maximum number of components in a given stage.

Efficiency Analysis

We begin by reviewing the standard framework for sequential algorithm analysis. We then consider the complications introduced by the introduction of parallelism and look at some proposed parallel frameworks. Analyzing Sequential Algorithms – The design and analysis of sequential algorithms is a well developed field, with a large body of commonly accepted results and techniques. This consensus is built upon the fact that the methodology and notation of asymptotic analysis (the so-called ―big-O notation) deliver results which are applicable across all sequential computers, programming languages, compilers and so on. This generality is achieved at the expense of a certain degree of blurring, in which constant factors and non-dominating terms in the analysis are simply ignored. In spite of this, the approach produces results which allow useful comparisons of the essential performance characteristics of different algorithms which are reflected in practice when implemented on real machines, in real languages through real compilers. For example, mergesort with its θ (n log n) run time is (in the worst case) an asymptotically better sorting algorithm than insertion sort (θ (n2) on any normal sequential machine (although the actual problem size at which the dominance becomes apparent will vary from implementation to implementation). Underpinning this work is the ―Random Access Machine(RAM) model which is an abstraction of the essential capabilities and cost characteristics which unite all sequential machines. The notation allows the description of

Upper bounds: Big Oh notation (O ()),

Lower bounds: Big Omega notation (Ω()) and

Tight bounds: Theta notation (θ())

on the behavior of functions representing the time or space requirements of an algorithm as its input problem size grows. Analyzing Parallel Algorithms – The sequential world benefits from a single universal abstract machine model (the RAM) which accurately (enough) characterizes all sequential computers and from a simple criterion of betterment for algorithm comparison (less is betterment, usually of run time, and occasionally of memory space). Thinking parallel, we immediately encounter two complications. Firstly, and fundamentally, there is no commonly agreed model of parallel computation.

The diversity of proposed and implemented parallel architectures is such that it is not clear that such a model will ever emerge. Worse than this, the variations in architecture capabilities and associated costs mean that no such model can emerge, unless we are prepared to forgo certain tricks or shortcuts exploitable on one machine but not another. An algorithm designed in some abstract model of parallelism may have asymptotically different performance on two different architectures (rather than just the varying constant factors of different sequential machines). Secondly, our notion of ―better‖ even in the context of a single architecture must surely take into account the number of processors involved, as well as the run time. The trade-offs here will need careful consideration. In this course we will not attempt to unify the irretrievably diverse. Thus we will have a small number of machine models and will design algorithms for our chosen problems for some or all of these. However, in doing so we still hope to emphasize common principles of design which transcend the differences in architecture. Equally, in some instances, we will exploit particular features of one model where that leads to a novel or particularly effective algorithm. Similarly, we will investigate notions of ―better‖ as they have been traditionally defined in the context of each model. We will continue to employ the notation o asymptotic analysis, but note that we must be particularly wary of constant factors in the parallel case – a ―constant factor‖ discrepancy of 32 in an asymptotically optimal algorithm on a 64 processor machine is a serious matter.