Parallel Algorithm Scalability Issues in Petaflops Architectures *

Ananth Grama[1], Anshul Gupta[2], Eui-Hong Han[3], and Vipin Kumar[3]

[1]Computer Science Department,
Purdue University,
West Lafayette, IN 47907.
ayg@cs.purdue.edu

[2]Mathematical Sciences Department,
IBM T.J. Watson Research Center, P.O. Box 218,
Yorktown Heights, NY 10598, USA.
anshul@watson.ibm.com

[3]Department of Computer Science,
University of Minnesota,
Minneapolis, MN 55455.
{kumar,han}@cs.umn.edu

Abstract

The projected design space of petaFLOPS architectures entails exploitation of very large degrees of concurrency, locality of data access, and tolerance to latency. This puts considerable pressure on the design of parallel algorithms capable of effectively utilizing increasing amounts of processing resources in a memory and bandwidth constrained environment. This aspect of algorithm design, also referred to as scalability analysis, is a key component for guiding algorithm designers as well as hardware architects. By quantifying the performance of an algorithm on larger machine configurations, scalability analysis guides parallel algorithm development. By identifying bottlenecks to scalability and machine parameters that influence these bottlenecks, scalability analysis influences hardware design. In this paper, we motivate the need for, and benefits of scalability analysis in the context of petaFLOPS systems. We present sample analyses

*This work is sponsored by NSF grant CCR-9972519, by Army Research Office grant DA/DAAG55-98-1-0441, and by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC and Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/ kumar.

of selected computational kernels from dense linear algebra, fast fourier transforms, and data intensive applications (association rule mining). The objective of this analysis is to demonstrate the analysis framework and its use in identifying desirable architectural features as well the ability of these selected kernels to scale to petaFLOPS systems.

1 Introduction

The impetus for petaFLOPS scale computing is provided by a number of critical applications from various domains [5, 32]. The target for achieving this scale of computing power in a 10-15 year time-frame will, in all likelihood, have to rely on large scale parallelism. Assuming a best-case scenario of exponential growth in peak processor speed, it is envisioned that petaFLOPS scale computers will require in the order of 10^4 computing elements to achieve required computation rates [32]. This projection is based on the assumption that processor speeds will increase to 100 GFLOPS in the projected timeframe. The degree of parallelism required for achieving petaFLOPS scale performance would be higher still if these processor speeds are not realized. For example, the first commercial PetaFLOPS computer to be announced, Blue Gene [10, 31], is expected to contain more than 1 million processors. This scale of parallelism puts severe strain on the ability of an algorithm to efficiently utilize all available resources. This key aspect of petaFLOPS scale computing accentuates the importance of the scalability of parallel algorithms; i.e., the ability of a parallel algorithm to yield good performance with increasing number of computing elements [11]. One of the fundamental challenges in petaFLOPS scale computing is to design the hardware platform and parallel algorithms for it in such a way that we can expect to get good performance on a very large number of processors.

The scalability of a parallel algorithm is limited by a variety of parallel computing overheads. The most commonly known is Amdahl's law, which states that if a fraction s of the total computation is serial, then the speedup of any parallel formulation is bounded by $1/s$. This simple rule has the consequence that if a computation has a serial component of 0.1%, it cannot use more than a thousand processors. Such computations can thus be ruled out as candidates for 10^4-fold parallelism. In general, the serial component of a computation is a function of the problem size itself. For example, the serial component of an n-integer addition is $\log n$. Given a certain number of processors on which to perform this computation, we would need to ensure that the reciprocal of the serial component exceeds the number of processors. This places constraints on the problem sizes that can be efficiently solved on large scale parallel platforms.

In addition to the serial component, a parallel computation incurs idling and communication overheads. The communication overhead is a function of the ability of the underlying interconnection network to deliver data between processing elements. Networking technology has seen significant improvements in the past few years. Data rates in the range of 1GB/s for point-to-point communications will be feasible in the foreseeable future on commercial platforms. This might give an impression that overheads associated with communication might become less significant as networking technology evolves. However, the scalability of an algorithm is often determined by the "balance factor" of a platform as opposed to raw communication latencies and bandwidths. Here, the "balance factor" loosely refers to the latency and bandwidth of the underlying network in relation to the computing speed of the processor. The balance factor of petaFLOPS scale platforms is unlikely to improve by over an order of magnitude and may even be worse than current generation systems. This implies that algorithmic innovations to minimize communication overheads will continue to be critical for the scalability of petaFLOPS scale computations.

The aforementioned motivations for scalability analysis have been known and understood for several years [14]. However, only a very small number of physical machines scaling up to several thousand processing elements were actually built. Most of these machines were built for very specific computations and the underlying algorithms were carefully crafted to utilize these machines. The only exceptions are the ASCI class machines and scalability issues are very relevant for these platforms as well. The petaFLOPS scale computing effort distinguishes itself from these both in terms of scale of platform as well as the diversity of application base. Specifically, we anticipate an order of magnitude increase in the number of processing elements and a much larger application base. This puts emphasis on i) designing platforms capable of sustained petaFLOPS scale performance across a wide range of applications; and ii) algorithm design methodologies and metrics that allow algorithms to use these platforms within the resource constraints of the platform. We explore these issues in the context of some commonly used computational kernels in this chapter.

The remainder of this chapter is organized as follows: Section 2 describes the terminology and definitions, and provides an overview of metrics for quantifying scalability; Section 3 presents case studies of scalability analysis of selected kernels from diverse domains and their architectural implications; Section 4 discusses extensions to the conventional scalability framework for petaFLOPS scale computing; and Section 5 presents concluding remarks on the need for scalability analysis.

2 Terminology, Definitions, and Background

In this section, we present terminology and definitions used through the rest of this paper. We will also lay a framework for an architecture independent analysis model and use it to motivate a number of scalability metrics. We conclude with a discussion of suitability of various metrics to petaFLOPS scale computers.

2.1 Basic Performance Metrics for Parallel Systems

The serial run time W of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The *parallel run time* T_P is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. When evaluating a parallel system, we are often interested in determining how much performance gain is achieved by parallelizing a given application over a sequential implementation. This measure, called *speedup*, captures the relative benefit of solving a problem on a parallel platform. Formally, speedup S is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors.

Due to various parallel processing overheads, processors in a parallel ensemble cannot devote all of their time to useful computation (computation that is also performed by the serial counterpart). The fraction of time spent by a processor on useful tasks determines the *efficiency*, E, of a parallel algorithm. Formally, efficiency is defined as the ratio of speedup to the number of processors. In the absence of superlinearity effects, the efficiency of a parallel program is bounded by one. Typical parallel programs incur overheads such as idling, interprocessor communication, and excess computation. These overheads can be encapsulated into a single function called the *overhead function*. We define *total overhead* or the *overhead function* of a parallel system as the part of its cost (processor-time product) that is not incurred by the fastest serial algorithm on a sequential computer. It is the total time collectively spent by all the processors over and above that spent in useful computation. We denote the overhead function of a parallel system by the symbol T_o. The overhead function depends on W and p, and we write it as $T_o(W, p)$.

The cost of solving a problem of size W on p processors, or the total time spent in solving a problem summed over all processors, is pT_P. W units of this time are spent performing useful work, and the remainder is overhead. Therefore, the relation between cost (pT_P), problem size (W), and the overhead function (T_o) is given by

$$T_o = pT_P - W. \tag{1}$$

Since the efficiency of a parallel program is given by

$$E = \frac{W}{pT_P}$$

Using the expression for parallel overhead (Equation 1), we can rewrite this expression as

$$E = \frac{1}{1 + \frac{T_o}{W}}. \tag{2}$$

Asymptotically, the total overhead function T_o is an increasing function of p. This is because every program must contain some serial component. If this serial component of the program takes time t_{serial}, then during this time, all the other processors must be idle. This corresponds to a total overhead function of $(p-1) \times t_{serial}$. Therefore, the total overhead function T_o grows at least linearly with p. Furthermore, due to communication, idling and excess computation, this function may grow superlinearly in the number of processors.

Equation 2 gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e. the value of W remains constant), as we increase the number of processors, T_o increases. In such a scenario, it is clear from Equation 2 that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processors for a given problem size is common to all parallel programs and is a major issue for scaling problems to petaFLOPS scale platforms.

On the other hand, increasing the problem size keeping the number of processors constant often has the opposite effect. We know that the total overhead function T_o is a function of both problem size W and the number of processors p. In many cases, T_o grows sub-linearly with respect to W. In such cases, from Equation 2, we can see that efficiency increases as the problem size is increased. This phenomenon is *not* true for all parallel programs. The class of programs for which this is true is referred to as *scalable*. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.

From Equation 2, the problem size W can usually be obtained as a function of p by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as p increases. We call this function the *isoefficiency function* of the parallel system. Here, a parallel system refers to the combination of the parallel architecture and the parallel algorithm.

Isoefficiency function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processors. A small isoefficiency function (for eg. $O(p)$, $O(p \log p)$, or $O(p^{1.5})$) means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. However, a high isoefficiency function (for eg. $O(p^3)$ or asymptotically higher) indicates a poorly scalable parallel system. For such systems, high efficiencies can be achieved on large numbers of processors only for excessively large problems. The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased. On the other hand, if a problem consists of W basic operations, then no more than W processors can be used to solve the problem in a cost-optimal fashion. This bound on the concurrency establishes a lower bound of $O(p)$ on the isoefficiency of any parallel system. Extensive analysis based on the isoefficiency function and detailed descriptions are provided in [27, 11].

2.1.1 Related Scalability Metrics

In addition to the isoefficiency metric, a number of other scalability metrics have also been proposed that are particularly suited to various scaling scenarios. Gustafson et al. [18, 16] introduced the scaled speedup metric, which is defined as the speedup obtained when the problem size is increased linearly with the number of processors. If the scaled-speedup curve is close to linear w.r.t. the number of processors, the parallel system is considered scalable. It is easy to see that a linear scaled speedup curve corresponds to a constant efficiency with linear problem scaling. This indicates a linear isoefficiency function. When the isoefficiency is superlinear (or non-existent), the scaled speedup curve is sublinear.

A number of other scalability metrics have also been proposed [35, 25, 36, 8, 26, 9, 33, 17, 34, 30, 8]. A detailed discussion of these and other metrics is provided in [14]. These scalability metrics are relevant to petaFLOPS scale computing to varying degrees. The isoefficiency metric is particularly useful because of the following features:

- It determines whether an algorithm can effectively utilize a given number of processors in the specified configuration for practical problem sizes. Here, practical problem sizes can be viewed in terms of the available memory, constraints on time to solution, or need for hardware resources such as communication bandwidth.

- It can be used to determine bottlenecks in the architecture that critically impact performance of a parallel algorithm and provides insights into alleviating these bottlenecks.

- It guides algorithm development by providing a framework for performance evaluation on point designs.

Memory Scalability Another important notion of scalability of a parallel algorithm is the requirement on aggregate memory imposed by the algorithm for efficient parallel execution. The key issue is how fast the aggregate memory (or memory per processor) should grow to allow efficient use of the parallel computer. As we have seen, an increase in the number of processors must be accompanied by an appropriate increase in the problem size to achieve good efficiency. This increase in problem size translates directly to an increase in the aggregate memory requirement of the parallel algorithm. If this rate of increase is superlinear, one can argue that the parallel algorithm will be unable to make use of a large number of processors. This is because one cannot expect the aggregate memory of the parallel computer to grow superlinearly in number of processors. Current generation of computers operate in the range of 1 Byte/FLOPS of memory. It is believed that this number is unlikely to increase on a petaFLOPS scale architecture (and is likely going to decrease because of cost and technology). This has important implications for architecture as well as algorithm design. Specifically, if problem scaling dictates superlinear aggregate memory growth, machine characteristics such as bandwidth and latency must be improved to compensate for it. It can be argued that aggregate memory must asymptotically scale at least linearly with the number of processors to maintain constant efficiency. This is because a sublinear memory growth indicates a reduction in the computation per processor with increasing p. This results in a lower computation-to-overhead ratio on a per-processor basis; and consequently a drop in efficiency.

Constraints on Time-to-Solution Increasing the problem size with number of processors to maintain fixed efficiency can lead to extremely large problem sizes. Consider two parallel systems – one with an $O(p)$ isoefficiency and one with an $O(p^2)$ isoefficiency. In the first case, a linear increase in problem size with number of processors is sufficient to maintain high efficiency. This implies that the time to solution remains approximately constant with increasing number of processors since the problem size per processor is constant. In the second case, the problem size per processor (and thus the approximate time to solution) increases linearly with the number of processors. This implies that if a problem taking 100 seconds on 4 processors yields an efficiency of 0.75, then on 10,000 processors to get an efficiency of 0.75, we would have to solve a problem whose parallel runtime on 10,000 processors is $100 \times 10000/4$, or approximately 3 days! If this problem was one of forecasting the next day's weather, the use of a petaFLOPS scale computer can be immediately ruled out using this algorithm-architecture combination. Such constraints on time-to-solution are particularly important for problems with hard deadlines and parallel systems with superlinear isoefficiencies.

3 Scalability Analysis of Algorithms

In this section, we demonstrate the power of scalability analysis using three commonly used parallel algorithms, namely, dense matrix multiplication, fast Fourier transform, and finding associations in data. We show how the scalability analysis can be used to identify desirable architectural features and performance bottlenecks for these algorithms. Between these three algorithms, a wide range of scalability issues are covered. Many parallel algorithms arising in a variety of applications share scalability issues with one of the three algorithms discussed in this section. For example, dense and sparse matrix factorization algorithms have scalability characteristics similar to those of dense matrix multiplication and the analysis of many algorithms for sorting and computing convolutions is similar to that of parallel FFT. For a detailed discussion of these algorithms, we refer the reader to [27].

3.1 A Model For Communication Costs in Parallel Programs

In order to analyze the selected kernels, we must first formalize a model for the interaction costs incurred by parallel programs. Interactions between processors can take the form of synchronizations or communications. While these interactions are explicit in message passing machines, they are implicit in shared address space platforms. In each case however, the associated time overhead is incurred by the parallel program. The time required for a single point-to-point communication over an uncongested interconnection network can be well approximated in terms of a startup latency t_s and a bandwidth-determined per-word transfer time t_w. Specifically, the time to communicate an m word message over an uncongested cut-though routed interconnection network can be approximated by:

$$T_{comm} = t_s + mt_w$$

For the purpose of analysis, it is often convenient to express t_s and t_w in terms of the time quanta for a unit computation as opposed to absolute time units. In the rest of this chapter, we will use this normalized representation of the startup and per-word transfer times.

Congestion on the underlying interconnection network may, however, increase this time. Communication patterns in various algorithms congest different architectures to varying degrees. The true communication time can be approximated as:

$$T_{comm} = t_s + mt_w f(A, C, p)$$

Here, function $f(A, C, p)$ is determined by the underlying interconnection network A, the communication pattern C, and the number of processing elements p linked by the network. In the absence of knowledge of the underlying interconnection architecture, we introduce the notion of effective word-transfer time t'_w as $t_w f(A, C, p)$. This abstraction has the advantage that we can analyze parallel algorithms in an architecture independent manner while making quantitative statements on architectural requirements of various algorithms. This is demonstrated in greater detail later in this section. Note that effective bandwidth can be defined only in the context of specific communication patterns and architectures, and depending on the communication pattern and architecture, it may or may not be a function of p.

3.2 Dense Matrix Multiplication

We introduce the use of the isoefficiency metric for scalability analysis using a simple parallel algorithm for dense matrix multiplication due to Cannon [7]. Although more efficient parallel algorithms for dense matrix multiplication exist [13], we chose to discuss Cannon's algorithms because of its simplicity.

Consider a logical two dimensional mesh of p processors (with $\sqrt{p}$ rows and $\sqrt{p}$ columns) on which two $n \times n$ matrices A and B are to be multiplied to yield the product matrix C. Let $n \geq \sqrt{p}$. The matrices are divided into sub-blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ which are mapped naturally on the processor array. The sub-blocks of A and B residing with the processor (i, j) are denoted by A^{ij} and B^{ij} respectively, where $0 \leq i < \sqrt{p}$ and $0 \leq j < \sqrt{p}$. In the first phase of the execution of the algorithm, the data in the two input matrices is aligned in such a manner that the corresponding square submatrices at each processor can be multiplied together locally. This is done by sending the block A^{ij} to processor $(i, (j+i) mod \sqrt{p})$, and the block B^{ij} to processor $((i+j) mod \sqrt{p}, j)$. The copied sub-blocks are then multiplied together.

Now the A sub-blocks are rolled one step to the left and the B sub-blocks are rolled one step upward and the newly copied sub-blocks are multiplied and the results added to the partial results in the C sub-blocks. Figure 1 illustrates the steps of the parallel algorithm.

3.2.1 Scalability Analysis

The multiplication of A and B is complete after $\sqrt{p}$ steps of rolling the sub-blocks of A and B leftwards and upwards, respectively, and multiplying the in coming sub-blocks in each processor. There are a total of $2\sqrt{p}$ message transfers of $\frac{n^2}{p}$ words of data each. The time spent in each communication step is $t_s + t_w\frac{n^2}{p}$. Therefore, the total parallel execution time is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}} \tag{3}$$

From Equation (3), it follows that the total overhead over all the processors for this algorithm is $2t_sp\sqrt{p} + 2t_wn^2\sqrt{p}$. In order to determine the isoefficiency term due to t_s, W has to be proportional to $2Kt_sp\sqrt{p}$, where $K = \frac{1}{1-E}$ and E is the desired efficiency that has to be maintained. Hence the following isoefficiency relation results:

$$n^3 = W \propto 2Kt_sp\sqrt{p} \tag{4}$$

Similarly, to determine the isoefficiency term due to t_w, n^3 has to proportional to $2Kt_wn^2\sqrt{p}$. Therefore,

$$\begin{aligned} n^3 &\propto 2Kt_wn^2\sqrt{p} \\ n^3 = W &\propto 8K^3t_w^3p^{1.5} \end{aligned} \tag{5}$$

According to both Equations (4) and (5), the asymptotic isoefficiency function of Cannon's algorithm is $O(p^{1.5})$.

Since the maximum number of processors that can be used by this algorithm is n^2, the isoefficiency due to concurrency is also $O(p^{1.5})$ ($n^2 \propto p \;=>\; n^3 = W \propto p^{1.5}$).

3.2.2 Scalability with Respect to Communication-Bandwidth and Memory

The preceding analysis clearly shows what rate the problem size needs to be scaled at with the number of processors to maintain a constant level of CPU utilization. However, it can also be used to derive a few other interesting conclusions regarding the behavior of Cannon's algorithm on a massively parallel architecture. First, since $W = n^3 \propto p^{1.5}$, therefore, $n^2 \propto p$. In other words, the memory required to solve the larger problems for maintaining a fixed efficiency grows only linearly with p. As a result, the algorithm-architecture combination can be scaled indefinitely without suffering a loss of efficiency as long as a constant amount of memory is added along with each processor. Secondly, note that a multiplicative factor of t_w^3 is associated with the isoefficiency term in Equation 5. As discussed earlier, t_w depends on the ratio of the data communication speed of the channels to the computation speed of the processors used in the parallel architecture. This means that if the processors of the multicomputer are replaced by k times faster processors while the interconnection bandwidth

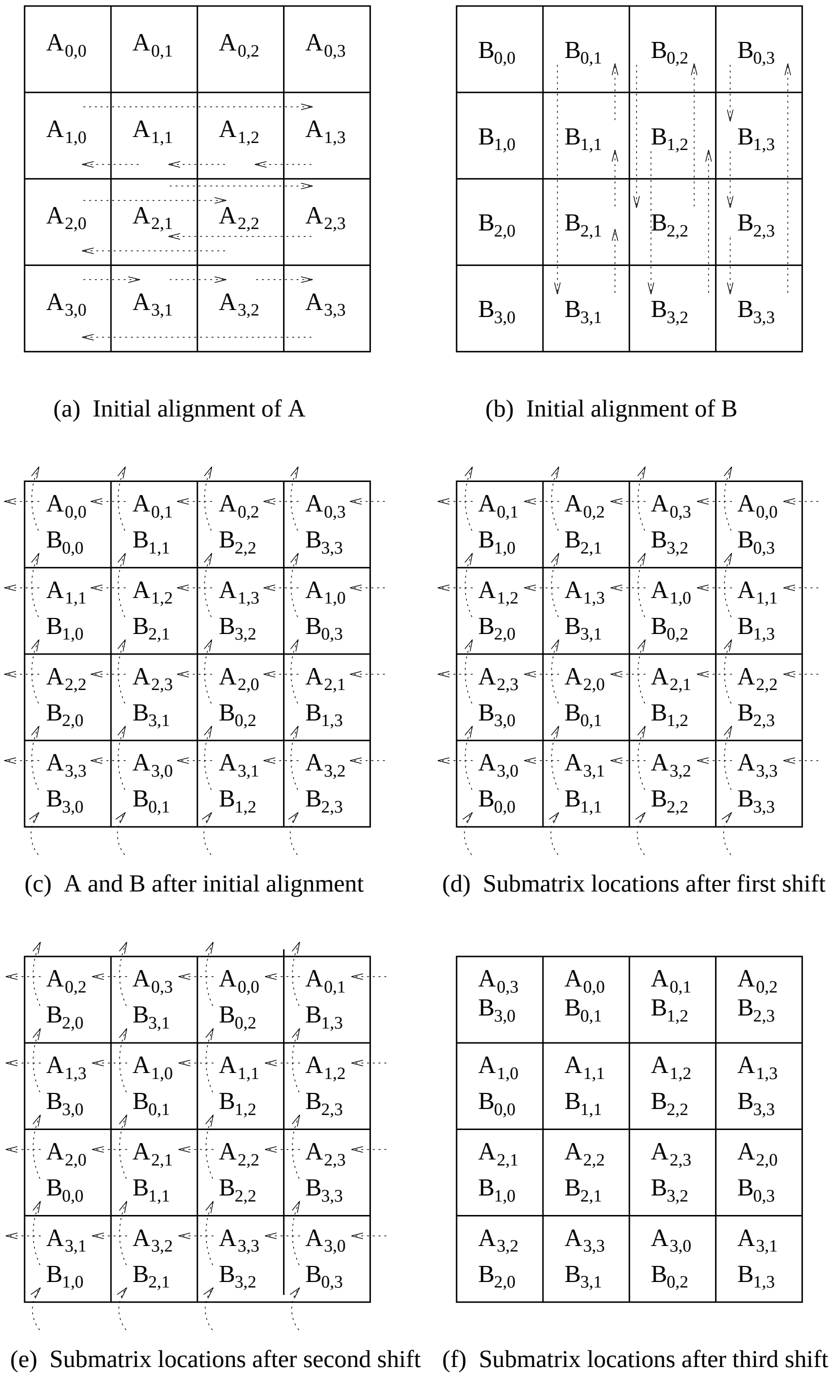

Figure 1: The communication steps in Cannon's algorithm on 16 processors.

remains the same, then the problem size will have to be increased by a factor of k^3 in order to obtain the same efficiency. On the other hand, increasing the number of processors by a factor of k requires the problem to be increased by only a factor of $k^{1.5}$ to maintain the same efficiency. This is contrary to the conventional wisdom that often suggests that better performance is always obtained using fewer faster processors [6].

3.2.3 Scalability of Some Related Matrix Algorithms

Dense matrix factorization Just like dense matrix multiplication, the best parallel algorithms for dense Cholesky and LU factorization without pivoting have isoefficiency functions of $O(p^{1.5})$ [27]. Since the computational complexity of these algorithms for $n \times n$ matrices (i.e., input size of $O(n^2)$) is $O(n^3)$, they scale with a constant memory use per processor. However, if numerical constraints require partial pivoting to be performed during LU factorization, then the resulting extra overhead due to the additional communication and constraints on pipelining increase the isoefficiency function to $O(p^{2.25})$. Therefore, in the presence of partial pivoting, the problem size cannot be increased indefinitely to maintain a fixed efficiency without eventually running out of memory.

Sparse matrix factorization Analyzing the scalability, or even the parallel run time, is much more involved in the case when the matrix being factored is sparse because the amounts of computation and overheads are very sensitive to the sparsity pattern of the original matrix. Such analysis has been performed for the class of sparse matrices that arise from the discretization of a physical domain. This analysis [12] reveals that by using the appropriate algorithms, an isoefficiency function as low as $O(p^{1.5})$ can be achieved for this class of sparse matrices (in the absence of partial pivoting). This is somewhat surprising because the amount of computation in sparse matrix factorization is much smaller compared to dense factorization, while the communication pattern is significantly more complex.

However, unlike dense factorization, sparse matrix factorization is not always scalable with respect to memory use. The memory requirement and the computational complexity of factoring a sparse matrix resulting from the discretization of a two-dimensional physical domain are $O(n \log n)$ and $O(n^{1.5})$, respectively. Since the isoefficiency function is $O(p^{1.5})$, the memory use will increase at a rate of $O(p \log p)$ if the problem size is increased at the rate of the isoefficiency function. Thus, parallel factorization is not scalable with respect to memory use in this case. For sparse matrices arising from three-dimensional domains, the space and time complexity of factorization is $O(n^{4/3})$ and $O(n^2)$, respectively. It is easy to determine that factoring such matrices is memory-scalable with an isoefficiency function of $O(p^{1.5})$.

3.3 Fast Fourier Transform

In this section, we analyze simplified versions[1] of two parallel FFT algorithms.

Figure 2 outlines the serial Cooley-Tukey algorithm for an n-point single dimensional unordered radix-2 FFT adapted from [4]. $\mathbf{X}$ is the input vector of length n ($n = 2^r$ for some integer r) and $\mathbf{Y}$ is its Fourier Transform. ω^k denotes the complex number $e^{j\frac{2\pi}{n}k}$, where $j = \sqrt{-1}$. More generally, ω is the primitive nth root of unity. Note that in the lth ($0 \leq l < r$)

[1] We do not take bit-reversal into account, which does not affect the asymptotic analysis.

1. **begin**
2. **for** i := 0 **to** n - 1 **do** R[i] := $\mathbf{X}_i$;
3. **for** l := 0 **to** r - 1 **do**
4. **begin**
5. **for** i := 0 **to** n - 1 **do** S[i] := R[i];
6. **for** i := 0 **to** n - 1 **do**
7. **begin**

(* Let $(b_0 b_1 \cdots b_{r-1})$ be the binary representation of i*)

8. $\mathrm{R}[(b_0 \cdots b_{r-1})] := \mathrm{S}[(b_0 \cdots b_{l-1} 0 b_{l+1} \cdots b_{r-1})] + \omega^{(b_l b_{l-1} \cdots b_0 0 \cdots 0)} \mathrm{S}[(b_0 \cdots b_{l-1} 1 b_{l+1} \cdots b_{r-1})]$;
9. **end**;
10. **end**;
11. **end**.

Figure 2: *The Cooley-Tukey algorithm for single dimensional unordered FFT.*

iteration of the loop starting on Line 3, those elements of the vector are combined whose indices differ by 2^{r-l-1}. Thus the pattern of the combination of these elements is identical to a butterfly network.

The computation of each $R[i]$ in Line 8 is independent for different values of i. Hence p processors ($p \leq n$) can be used to compute the n values on Line 8 such that each processor computes $\frac{n}{p}$ values. For the sake of simplicity, assume that p is a power of 2, or more precisely, $p = 2^d$ for some integer d such that $d \leq r$. To obtain good performance on a parallel machine, it is important to distribute the elements of vectors R and S among the processors in a way that keeps the interprocess communication to a minimum. As discussed in Section 2, there are two main contributors to the data communication cost - the message startup time t_s and the per-word transfer time t_w. In the following subsections, we present two parallel formulations of the Cooley-Tukey algorithm. As the analysis of Sections 3.3.1 and 3.3.2 will show, each of these formulations minimizes the cost due to one of these constants.

3.3.1 The Parallel Binary-exchange Algorithm

In the most commonly used mapping that minimizes communication for the binary-exchange algorithm [15, 23, 27, 28, 29], if $(b_0 b_1 \cdots b_{r-1})$ is the binary representation of i, then for all i, $R[i]$ and $S[i]$ are mapped to processor number $(b_0 \cdots b_{d-1})$.

With this mapping, processors need to communicate with each other in the first d iterations of the main loop (starting at line 3) of the algorithm. For the remaining $r - d$ iterations of the loop, the elements to be combined are available on the same processor. Also, in the lth $(0 \leq l < d)$ iteration, all the $\frac{n}{p}$ values required by a processor are available to it from a single processor; *i.e.*, the one whose number differs from it in the lth most significant bit.

Several factors may contribute to T_o in a parallel implementation of FFT [15]. The most significant of these overheads is due to data communication between processors. As discussed in Section 3.3.1, the p processors communicate in pairs in d $(d = \log p)$ of the r $(r = \log n)$

iterations of the loop starting on Line 3 of Figure 2. Since each processor stores and transfers $\frac{n}{p}$ words in each of the d communication steps, the total communication cost on p processors is:

$$T_o = t_s p + t_w n \log p \tag{6}$$

If p increases, then in order to maintain the efficiency at some value E, W should be equal to KT_o, where $K = E/(1-E)$. Since $W = n \log n$, n must grow such that

$$n \log n = K(t_s p \log p + t_w n \log p) \tag{7}$$

Clearly, the isoefficiency function due to the first term in T_o, is given by:

$$W \propto K t_s p \log p \tag{8}$$

The requirement on the growth of W (to maintain a fixed efficiency) due to the second term in T_o is more complicated. If this term requires W to grow at a rate less than $O(p \log p)$, then it can be ignored in favor of the first term. On the other hand, if this term requires W to grow at a rate higher than $O(p \log p)$, then the first term of T_o can be ignored.

Balancing W against the second term only yields the following:

$$\begin{aligned} n \log n &\propto K t_w n \log p \\ n &\propto p^{K t_w} \end{aligned} \tag{9}$$

This leads to the following isoefficiency function (due to the second term of T_o):

$$W \propto K t_w \times p^{K t_w} \times \log p \tag{10}$$

This growth is less than $O(p \log p)$ as long as $K t_w < 1$. As soon as this product exceeds 1, the overall isoefficiency function is given by Equation (10). The isoefficiency function given by Equation (10) deteriorates rapidly with the increase in the value of $K t_w$. In fact, the efficiency corresponding to $K t_w = 1$, (*i.e.*, $E = 1/(1 + t_w)$) acts somewhat as a threshold value. For a given parallel computer with a fixed t_w, efficiencies up to this values can be obtained easily. But efficiencies much higher than this threshold can be obtained only if the problem size is extremely large. The following examples illustrate the effect of the value of $K t_w$ on the isoefficiency function.

Consider a hypothetical parallel computer for which the relative values of the hardware parameters are given by $t_w = 2$, and $t_s = 12$. With these values, the threshold efficiency $1/(1 + t_w)$ is 0.33.

The isoefficiency function of this algorithm due to concurrency is $p \log p$. The isoefficiency function due to the t_s and t_w terms in the overhead function are $K t_s p \log p$ and $K t_w p^{K t_w} \log p$, respectively. To maintain a given efficiency E (that is, for a given K), the overall isoefficiency function is given by:

$$W \propto \max\{K t_s p \log p, K t_w p^{K t_w} \log p\} \tag{11}$$

Figure 3 shows the isoefficiency curves given by this function for $E = 0.20$, 0.25, 0.30, 0.35, 0.40, and 0.45. Notice that the various isoefficiency curves are regularly spaced for

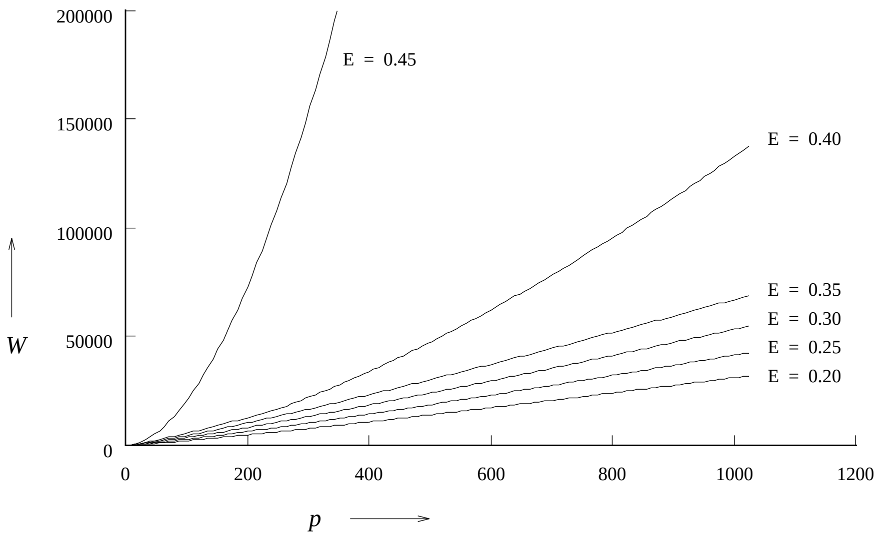

Figure 3: Isoefficiency functions of the binary-exchange algorithm on a computer with $t_w = 2$ and $t_s = 12$ for various values of E.

efficiencies up to the threshold. However, the problem sizes required to maintain efficiencies above the threshold are much larger. The asymptotic isoefficiency functions for $E = 0.20$, 0.25, and 0.30 are $O(p \log p)$. The isoefficiency function for $E = 0.40$ is $O(p^{1.33} \log p)$, and that for $E = 0.45$ is $O(p^{1.64} \log p)$.

Figure 4 shows the efficiency curve of n-point FFTs on a 256-processor hypercube with the same hardware parameters. The figure shows that the efficiency initially increases rapidly with the problem size, but the efficiency curve flattens out beyond the threshold.

The above examples show that there is a limit on the efficiency that can be obtained for reasonable problem sizes, and that the limit is determined by the ratio between the CPU speed and the effective bandwidth of the communication channels of the parallel computer. This limit can be raised by increasing the bandwidth of the communication channels. However, making the CPUs faster without increasing the communication bandwidth lowers the limit. Hence, the binary-exchange algorithm performs poorly on a computer whose communication and computation speeds are not balanced. If the hardware is balanced with respect to its communication and computation speeds, then the binary-exchange algorithm is fairly scalable, and reasonable efficiencies can be maintained while increasing the problem size at the rate of $O(p \log p)$.

3.3.2 The Parallel Transpose Algorithm

Let the vector $\mathbf{X}$ be arranged in an $\sqrt{n} \times \sqrt{n}$ two dimensional array in row major order. An unordered Fourier Transform of $\mathbf{X}$ can be obtained by performing an unordered radix-2 FFT over all the rows of this 2-D array followed by an unordered radix-2 FFT over all the columns. The row FFT corresponds to the first $\frac{\log n}{2}$ iterations of the FFT over the entire vector $\mathbf{X}$ and the column FFT corresponds to the remaining $\frac{\log n}{2}$ iterations. In a parallel implementation, this $\sqrt{n} \times \sqrt{n}$ can be mapped on to p processors ($p \leq \sqrt{n}$) such that each processor stores $\frac{\sqrt{n}}{p}$ rows of the array. Now the FFT over the rows can be performed without

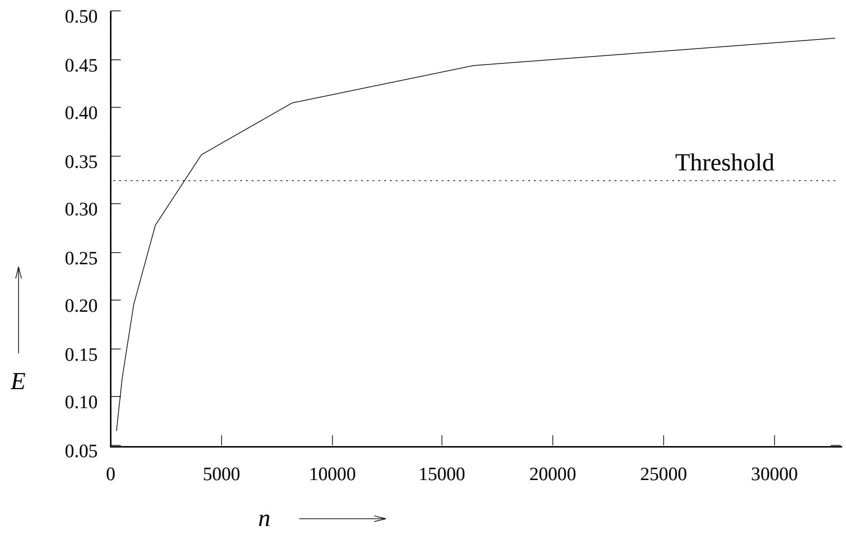

Figure 4: The efficiency of the binary-exchange algorithm as a function of n on a 256-processor parallel computer with $t_w = 2$, and $t_s = 12$.

any inter-processor communication. After this step, the 2-D array is transposed and an FFT of all the rows of the transpose is computed. The only step that requires any inter-processor communication is transposing an $\sqrt{n} \times \sqrt{n}$ array on p processors.

The only data communication involved in the transpose algorithm is the transposition of an $\sqrt{n} \times \sqrt{n}$ two dimensional array on p processors. This involves the communication of a unique block of data of size $\frac{n}{p^2}$ between every pair of processors. This communication (known as *all-to-all personalized communication*) can be performed by executing the following code on each processor:

for $i = 1$ **to** p **do**
  send data to processor number $(\textit{self\_address} \oplus i)$

This communication step takes $t_w \frac{n}{p} + t_s p$ time and yields the following overhead function:

$$T_o = t_w n + t_s p^2 \tag{12}$$

The first term of T_o is independent of p and hence, as p increases, the problem size must increase to balance the second communication term. For an efficiency E, this yields the following isoefficiency function, where $K = \frac{E}{1-E}$:

$$W \propto K t_s p^2 \tag{13}$$

The isoefficiency function of the transpose algorithm for FFT given by Equation 13 is a polynomial function of p and does not change asymptotically with hardware related parameters. This is in contrast to the isoefficiency function for the binary-exchange algorithm given by Equation 11, which changes asymptotically depending on t_w (i.e., the ratio of the effective bandwidth to the computation speed of each CPU). Therefore, for the range of t_w in which $Kt_w < 2$, the binary exchange algorithm is preferable. On the other hand, for parallel computers on which the communication speed is not balanced with respect to the computation speed, the transpose algorithm is more suitable. The transpose algorithm does not have a t_w-dependent efficiency threshold.

3.3.3 Impact of Architecture on the Effective t_w

Usually, the per-word communication time t_w is a function of the bandwidth of the links in the communication network and the speed and number of buffer copies involved in passing a message. However, for some communication patterns on sparse networks (i.e., networks whose bisection bandwidth is less than $\Omega(p)$), t_w can be a diminishing function of the number of processors p. The communication pattern of both the binary-exchange and the transpose algorithms is such that congestion in a sparse network will result in a t_w whose effective value is higher than the per word transfer time of a point-to-point link between two processors in the network. Ignoring the component of t_w due to message-buffer copying operations, the effective t_w will be higher than the per-word link transfer time by a factor that is the ratio of $p/2$ to the actual bisection width of the sparse network. On a dense network like the hypercube, this ratio is 1, but on a 2-D mesh, this ratio is $\sqrt{p}/2$. Therefore, the asymptotic isoefficiency function of the FFT algorithm will be higher on a sparse network than predicted by Equations 10 and 13.

3.3.4 Scalability of Parallel FFT with Respect to Memory Use

For the range of efficiency and t_w in which the asymptotic isoefficiency function of the binary-exchange FFT algorithm is $O(p \log p)$ (i.e., $\frac{Et_w}{1-E} \leq 1$), the $O(n)$ memory requirement for the FFT is proportional to p. However, if the asymptotic isoefficiency function is greater than $O(p \log p)$, which is the case for the transpose algorithm or for the binary-exchange algorithm when $\frac{Et_w}{1-E} > 1$, then maintaining a fixed efficiency will require an increasing amount of memory per processor as the number of processors increase. The isoefficiency function of the transpose algorithm is always $O(p^2)$. Therefore, the transpose algorithm is not scalable with respect to memory consumption.

3.4 Parallel Algorithms for Discovering Associations in Transaction Data

Consider the problem of discovering associations present in the data. The problem was formulated originally in the context of mining transaction data at supermarket, and since has been found to be applicable to a wide variety of domains such as data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket. The *market basket* data, as it is popularly known, consists of transactions made by each customer. Each transaction contains items bought by the customer. The goal is to see if occurrence of certain items in a transaction can be used to deduce occurrence of other items, or in other words, to find associative relationships between items [1]. This problem is far from trivial because of the exponential number of ways in which items can be grouped together. Hence, much research effort has been put into devising effective serial and parallel algorithms to solve this problem. A comparative survey of many parallel algorithms for finding associations is given in [24].

The computational core of association discovery is the phase of computing the frequency of a set of itemsets in a transaction database. We are given a set T containing n transactions and a set I containing m itemsets. Each transaction and itemset contains a small number of items, out of a possible set of items M. Our goal is to find the number of times that each itemset in I appears in each transaction. Figure 5(a) shows an example of this type of computation. Our database consists of ten transactions, and we are interested in computing the frequency of the eight itemsets shown in the second column. The actual frequencies of these itemsets in the database are shown in the third column. For instance, itemset {D, K} appears two times, once in the second and once in the ninth transaction. The computation of itemset frequency is optimized by storing the itemsets in a hash tree [3]. The run time of the serial algorithm for processing n transactions with m possible frequent item sets is:

$$\begin{aligned} T_{serial} &= \underbrace{n \times T_{trans}}_{\text{frequency counting}} + \underbrace{O(m)}_{\text{hash tree construction}} \\ &= O(n) + O(m) \end{aligned} \qquad (14)$$

where T_{trans} is the cost of updating itemset frequency in a hash tree per transaction.

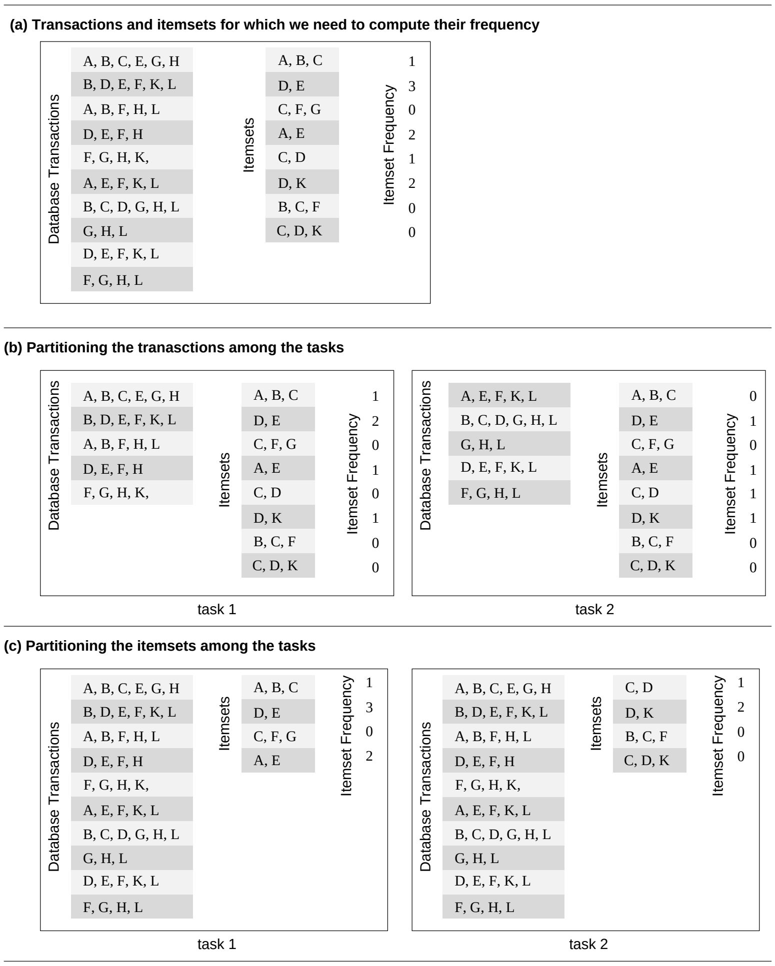

Figure 5: A sample transaction database and itemsets.

In this problem, there are two inputs, the sets T and I and one output, an array of size m that stores the frequency of each itemset in I. Any of these data could be partitioned to induce a decomposition of the overall computations for parallel processing. One possibility is to partition the set of transactions T into equal size parts, as shown in Figure 5(b). Another possibility is to partition the itemsets I, into an equal number of parts, as shown in Figure 5(c).

The *CD* algorithm [2] is an example of the first approach that partitions the set of transactions T across the processors. Task associated with each processor now is responsible for computing the frequencies of all the itemsets in I based solely upon the transactions it owns. The final result is obtained by adding these frequencies over all the processors. The runtime of the *CD* algorithm with p processors is [19]:

$$\begin{aligned} T_{CD} &= \underbrace{\frac{n}{p} \times T_{trans}}_{\text{frequency counting}} + \underbrace{O(m)}_{\text{global reduction}} + \underbrace{O(m)}_{\text{hash tree construction}} \\ &= O(\frac{n}{p}) + O(m) \end{aligned} \tag{15}$$

Comparing Equation 15 to Equation 14, we see that *CD* is able to perfectly parallelize the computation for frequency counting, except that it adds an addition step of global reduction that takes $O(m)$ time. However, the step of the hash tree construction is completely serial. Hence, a constant efficiency can be maintained as long as the number of transactions increases linearly with the number of processors while the size of the hash tree remains constant. However, if the problem size grows due to the increase in the size of hash tree (this can happen if we desire frequent itemsets of smaller support threshold), then *CD* does not scale at all.

The *IDD* [19] algorithm is an example of the second approach that partitions the itemsets I. The key feature is to distribute the itemsets to processors so as to extract the concurrency in candidate generation as well as counting phases. Task associated with each processor now is to compute the frequencies of only the itemsets that it owns. *IDD* employs various ways to reduce the communication overhead, to exploit the total available memory, and to achieve reasonable load balance [19]. The runtime of the *IDD* algorithm is [19]:

$$\begin{aligned} T_{IDD} &= \underbrace{n \times \frac{T_{trans}}{p}}_{\text{frequency counting}} + \underbrace{O(n)}_{\text{data movement}} + \underbrace{O(\frac{m}{p})}_{\text{hash tree construction}} \\ &= O(n) + O(\frac{m}{p}) \end{aligned} \tag{16}$$

Comparing Equation 16 to Equation 14, we see that *IDD* has a better scalability than *CD* with respect to the hash tree size, m. However, the cost of data movement in *IDD* is $O(n)$ and thus *IDD* is not scalable with respect to n.

Formulations that combine the approaches of replicating and distributing candidates so as to reduce the problems of each one, have been developed. An example is the *HD* algorithm of [19]. Briefly, it works as follows. Consider a p-processor system in which the processors are split into g equal size groups, each containing p/g processors. In the *HD* algorithm, we execute the *CD* algorithm as if there were only p/g processors. That is, we partition the transactions of the database into p/g parts each of size $n/(p/g)$, and assign the task of computing the counts of the itemset I for each subset of the transactions to each one of these groups of processors. Within each group, these counts are computed using the *IDD* algorithm. The *HD* algorithm inherits all the good features of the *IDD* algorithm. It also provides good load balance and enough computation work by maintaining minimum number of candidates per processor. At the same time, the amount of data movement in this algorithm is cut down to g/p of that of *IDD*. The runtime of *HD* is [19]:

$$\begin{aligned} T_{HD} &= \underbrace{\frac{g \times n}{p} \times T_{trans}}_{\text{frequency counting}} + \underbrace{O(\frac{m}{g})}_{\text{global reduction}} + \underbrace{O(\frac{g \times n}{p})}_{\text{data movement}} + \underbrace{O(\frac{m}{g})}_{\text{hash tree construction}} \\ &= O(\frac{g \times n}{p}) + O(\frac{m}{g}) \end{aligned} \quad (17)$$

Compared to the serial runtime, Equation 17 shows that *HD* is scalable with respect to m just like *IDD*. *HD* scales with increasing m provided g is chosen such that $\frac{m}{g}$ is constant. *HD* also has data movement cost. However, when p is increased with increasing n, the cost is almost constant provided g is unchanged. Thus *HD* is scalable with respect to increasing n. When n and m increase at the same time, *HD* is unscalable because g needs to be fixed with respect to increasing n and grow linearly with respect to increasing m. In conclusion, massively parallel computers can be used to find patterns in ever larger data sets or ever finer patterns, but not both. A detailed parallel runtime analysis of *HD* can be found in [19].

4 Extensions to Scalability Analysis Frameworks

While isoefficiency and related scalability metrics have been used for developing a variety of parallel algorithms, the underlying cost model has a critical impact on the accuracy of analytical results. In our examples in Section 3, the underlying cost model relied on a latency-bandwidth framework for point-to-point communication. While this framework is a good approximation of well calibrated and largely static message passing networks, it is likely that such a clean communication abstraction will not be accurate for a petaFLOPS scale architecture.

Modeling shared address spaces, impact of multithreading, and hardware assisted asynchronous communications pose challenges for communication abstraction. This is largely due to the fact that overheads in these scenarios are dependent not just on hardware parameters but also on scheduling mechanisms. For example, two thread schedules may result in markedly different cache miss patterns or false sharing overheads. While some work has been done on modeling contention [21, 20, 22], accurate abstractions of such platforms are difficult.

Conventional processors rely on multiple levels of caches to mask memory latency and bandwidth bottlenecks. This poses challenges not just for optimizing programs for optimal cache performance but also modeling program performance. It is realistic to assume that this hierarchy will get deeper in petaFLOPS scale architectures; thus compounding issues program performance modeling for deep hierarchies.

Finally, the programming model for petaFLOPS scale machines might be a mix of shared address space and message passing paradigms. To tolerate high latencies of communication, one might have to rely on threaded message passing programs in which threads swap out on a communication operation and become ready when the operation has been completed.

Modeling such mixed-mode programs involves the complexities of both shared address space and message passing paradigms.

5 Concluding Remarks

As we have demonstrated in this chapter, scalability analysis can play a critical role in the development of petaFLOPS class systems. It gives a clear view of constraints on the hardware architecture that must be satisfied for scalability to large platforms. As examples, we demonstrate the following interesting aspects with respect to the algorithms analyzed:

- We show that achieving good efficiencies on dense matrix multiplication only requires a linear scaling of memory with number of processors. As we have seen, this is an ideal memory scaling scenario. We also demonstrate the impact of the balance factor for dense matrix multiplication. Specifically, we show that if the processor in a petaFLOPS system is replaced by one k times faster without changing the interconnect, the problem size will have to be scaled up by a factor of k^2 to achieve the same efficiency. This quadratic scaling implies that the algorithm is very sensitive to the balance factor of the machine. On the other hand, increasing the number of processors by a factor of k only requires the problem size to be increased by a factor of $k^{1.5}$ to maintain efficiency. Clearly, the latter is a more desirable scaling scenario and yields insights into desirable parts of the design space. This is particularly relevant for efforts focused on using powerful superconducting processors as building blocks for petaFLOPS computers.

- The transpose-based parallel FFT algorithm, which is well known to be communication efficient is unscalable with respect to memory use.

- For FFTs, we also demonstrate that sparse interconnects can result in very poor isoefficiency functions. For such scenarios, it is very difficult to achieve good performance on larger number of processors. One solution is to increase the effective bandwidth of sparse networks to compensate for network congestion. Maintaining good isoefficiency is critical for the memory scalability of the algorithm as well. We show that the memory requirement is linear in number of processors only when the isoefficiency function is $O(p \log p)$. Thus, if FFT is a target kernel for the petaFLOPS machine, bisection width of the interconnect is a critical consideration, both for efficient execution and memory scalability.

- In the context of finding associations in transaction datasets, we show that petaFLOPS-scale parallel computers can be used for finding patterns in ever larger datasets or ever-finer patterns, but not both.

Similar analysis can be performed for other kernels of interest as well. The isoefficiency framework is demonstrated in the context of other algorithms in [27, 11]. Its relationship to other scalability metrics is explored in [14].

Just as scalability analysis of an algorithm directs hardware design, hardware characteristics can also be used to identify algorithms that can effectively use a given platform. This is important because point designs can be evaluated for their ability to support target computational kernels before assembling them. In this manner, scalability analysis provides an effective information conduit between hardware-algorithm co-design for petaFLOPS scale systems.

Finally, in many applications, there is a choice of kernels that can be used to solve a given problem. While in the serial context, the tradeoffs among kernels can be made in terms of their memory and computational characteristics (complexities for example), in a large-scale parallel context, one must also consider the scalability of these kernels. For example, a block dense formulation of some sparse problems might have a higher serial complexity but might scale better to a petaFLOPS scale system. The net effect would be that a larger problem can sometime be solved effectively using a kernel that is inferior in the serial context. In this manner, scalability analysis can aid kernel selection for many applications.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of 1993 ACM-SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[2] R. Agrawal and J.C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Eng.*, 8(6):962–969, December 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[4] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[5] David Bailey. The 1997 Petaflops Algorithms Workshop Summary Report. URL: `http://www.ccic.gov/cicrd/pca-wg/pal97.html`, 1997.

[6] M. L. Barton and G. R. Withers. Computing performance as a function of the speed, quantity, and the cost of processors. In *Supercomputing '89 Proceedings*, pages 759–764, 1989.

[7] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozman, MT, 1969.

[8] S. Chandran and Larry S. Davis. An approach to parallel vision algorithms. In R. Porth, editor, *Parallel Processing.* SIAM, Philadelphia, PA, 1987.

[9] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[10] Linda Geppert. Microprocessors: The off-beat generation. *IEEE Spectrum*, 37(7):44–49, July 2000.

[11] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August, 1993.

[12] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[13] Anshul Gupta and Vipin Kumar. The scalability of matrix multiplication algorithms on parallel computers. Technical Report TR 91-54, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1991. A short version appears in *Proceedings of 1993 International Conference on Parallel Processing*, pages III-115–III-119, 1993.

[14] Anshul Gupta and Vipin Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19:234–244, 1993. Also available as Technical Report TR 92-32, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[15] Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993. A detailed version available as Technical Report TR 90-53, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[16] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[17] John L. Gustafson. The consequences of fixed time performance measurement. In *Proceedings of the 25th Hawaii International Conference on System Sciences: Volume III*, pages 113–124, 1992.

[18] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[19] E.-H. Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Eng.*, 12(3), May/June 2000.

[20] Chris Holt, Jaswinder Pal Singh, and John Hennessy. Application and architectural bottlenecks in distributed shared memory multiprocessors. In *In Proceedings of the 22nd Intl. Symposium on Computer Architecture (ISCA)*, May 1996.

[21] Matthew I.Frank, Anant Agarwal, and Mary K.Vernon. Lopc: Modeling contention in parallel algorithms. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 276–287, Las Vegas, NV, June 18 - 21, 1997.

[22] Dongming Jiang and Jaswinder Pal Singh. Scaling application performance on cache-coherent multiprocessors. In *Proc. International Symposium on Computer Architecture*, May 1999.

[23] S. L. Johnsson, R. Krawitz, R. Frye, and D. McDonald. A radix-2 FFT on the connection machine. Technical report, Thinking Machines Corporation, Cambridge, MA, 1989.

[24] Mahesh V. Joshi, E.-H. Han, George Karypis, and Vipin Kumar. Efficient parallel algorithms for mining associations. In M. J. Zaki and C.-T. Ho, editors, *Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence (LNCS/LNAI)*, volume 1759. Springer-Verlag, To Appear.

[25] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[26] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC13572, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[27] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, 1994.

[28] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, PA, 1992.

[29] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared memory architectures. *IEEE Transactions on Computers*, C-36(5):581–591, 1987.

[30] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):57–61, 1991.

[31] Robert Pool. Assembling life's building blocks. *IBM Research Magazine*, (4), November 1999. URL: http://www.research.ibm.com/resources/magazine/1999/number_4/bluegene499.html.

[32] Thomas Sterling, Paul Messina, and Paul H. Smith. *Enabling Technologies for Petaflops Computing.* MIT Press, 1995.

[33] Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, December 1991. Also available as Technical Report IS-5053, UC-32, Ames Laboratory, Iowa State University, Ames, IA.

[34] Xian-He Sun and Diane Thiede Rover. Scalability of parallel algorithm-machine combinations. Technical Report IS-5057, Ames Laboratory, Iowa State University, Ames, IA, 1991. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

[35] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990.

[36] J. R. Zorbas, D. J. Reble, and R. E. VanKooten. Measuring the scalability of parallel computer systems. In *Supercomputing '89 Proceedings*, pages 832–841, 1989.