On the Sublinear Processor Gap for Parallel Architectures

Alejandro López-Ortiz and Alejandro Salinger

David R. Cheriton School of Computer Science, University of Waterloo
{alopez-o,ajsalinger}@uwaterloo.ca

Abstract. In the past, parallel algorithms were developed, for the most part, under the assumption that the number of processors is $\Theta(n)$ (where n is the size of the input) and that if in practice the actual number was smaller, this could be resolved using Brent's Lemma to simulate the highly parallel solution on a lower-degree parallel architecture. In this paper, however, we argue that design and implementation issues of algorithms and architectures are significantly different—both in theory and in practice—between computational models with high and low degrees of parallelism.

We report an observed gap in the behavior of a parallel architecture depending on the number of processors. This gap appears repeatedly in both empirical cases, when studying practical aspects of architecture design and program implementation as well as in theoretical instances when studying the behaviour of various parallel algorithms. It separates the performance, design and analysis of systems with a sublinear number of processors and systems with linearly many processors. More specifically we observe that systems with either logarithmically many cores or with $O(n^\alpha)$ cores (with $\alpha < 1$) exhibit a qualitatively different behavior than a system with a linear number of cores on the size of the input, i.e., $\Theta(n)$. The evidence we present suggests the existence of a sharp theoretical gap between the classes of problems that can be efficiently parallelized with $o(n)$ processors and with $\Theta(n)$ processors unless $P = NC$.

1 Introduction

There is a vast experience in the study and development of algorithms for the PRAM architecture. In this case, the standard assumption (though often unstated) was that the number of processors p was linear on the size of the input, i.e., $p = O(n)$ (see for example [18] for a thorough discussion). Indeed, the definition of the class NC, which is often equated with the class of problems that can be efficiently parallelized on a PRAM, allows for up to polynomially many processors. Hence algorithms were designed to handle the case when $p = \Theta(n)$ or $p = \Theta(n^k)$ for $k \geq 1$ and if the actual number of processors available was lower, this could readily be handled by Brent's Lemma using a suitable scheduler [10, 5]. A fruitful theory was developed under these assumptions, and papers in which $p = o(n)$ were relatively rare.

In this paper, we analyze and report on the influence of the assumed number of processors on several aspects of the performance of various types of parallel architectures. Because of its current prevalence, we focus especially in multi-core architectures, which actually feature a relatively small number of processors and hence advantages that can be identified for parallel systems with a small number of processor count can lead to benefits in parallel computation in these architectures. However, we also report on aspects of computation that are relevant in general in other architectures such as memory collisions, communication in distributed architectures, and network sizes, as well as in more theoretical aspects like complexity classes and simulations of other models. Our observations suggest the existence of fundamental differences in the qualities of parallel systems with sublinear and linear number of processors, and that exploiting the advantages of the former can lead to more

Proc. count	$\Theta(n)$	$\Theta(n^\alpha)$	$\Theta(\log n)$
Merge sort	N	N	Y
Master theorem			
-Case 1	N	Y	Y
-Case 2	N	Y	Y
-Case 3	N	N	N
Amdahl's law	N	1/2	Y
Collision	N	Y	Y
Buffering	N	N	Y
Network size	N	1/2	Y
TM simulation	N	N	Y

Table 1. Optimal performance for each case according to processor count.

practical and conceptually simpler designs of both parallel architectures and algorithms, ultimately increasing their adoption and reducing development costs.

2 Overview of arguments

In this section we briefly list the arguments in favour of considering a limited degree of parallelism. We emphasize that we did not start from the outset with this goal, but rather we sought to develop algorithms and tools (both practical and theoretical) for current multi-core architectures. The observations within are derived from both theoretical investigations and practical experiences in which time and time again we found that there seems to be a qualitative difference between a model with $O(\log(n))$ processors and one with $O(n)$ processors, with, surprisingly, the advantage being for the weaker, i.e., $O(\log(n))$ model. Table 1 shows a summary of our observations for the considered processor counts. There is strong evidence of a sublinear cliff, beyond which development and implementation of efficient PRAM algorithms for many problems is substantially harder if not completely impossible, unless $P = NC$. In several instances among the evidence observed, the phenomenon had been observed earlier by others [18, 20, 14]. We now list our arguments briefly, before we expand on each of them individually in the next section.

1. The number of cores in current multi-core processors is nearly a constant, but first, if it is truly a constant there is nothing we can say, and second, it seems to be steadily though slowly growing.
2. In analogous fashion to the word-RAM, the number of bits in a word could be an arbitrary w but really it is most likely $\Theta(\log n)$ since it is also an index into memory and memory is usually polynomial on n.
3. The probability of collision on a memory access is only acceptably low for up to $O(\sqrt{n})$ processors.
4. The number of interconnects on a CPU network grows too fast for anything else.
5. Serialization at the network end is too costly, i.e., if more than two processors want to talk to you at the same time you have to listen to them serially.
6. There are natural $\log n$ and n^ϵ barriers in the complexity of designing algorithms.
7. Efficient cache performance requires bounded number of processors in terms of cache sizes, which are always assumed to be below n, and often as well in terms of the ratio of shared and private cache sizes, which is well below 100.

8. We define the class of problems which can be sped up using a logarithmic number of processors and show that it contains ENC and EP [20] and furthermore, this containment is strict.
9. For Turing Machines we can automatically increase performance by a $\log n + \log\log n$ factor when simulating with a parallel computer and this works for $\log n$ processors.
10. Amdahl's law suggests that programs can only noticeably benefit from parallelism if the number of processors is proportional to the relative difference between the execution time of the serial and parallel portions of a program.

3 Exposition

In this section we briefly expand on each of the points above. We aim to keep each argument as short as possible, since the entirety of the case is more important than any individual point.

3.1 Limited Parallelism

In principle it is possible to build a computer with an arbitrary degree of parallelism. In practice PRAMs algorithms and architectures focused on $\Theta(n)$-processor architectures, while relying on Brent's Lemma for cases when the number of processors was below that. In contrast multi-core processors have aimed for a much smaller number of cores. In principle this number could be modeled as a constant. However this is unrealistic as the number of cores continues to grow—albeit slowly—with desktop computers having transitioned over the last decade from single core to dual core to quad core and presently eight cores and sixteen cores already shipping at the higher end of the spectrum. Additionally, it has been observed that generally speaking larger inputs justify larger investments in RAM and CPU capacity, so a function of n is much more reflective of real life constraints. This suggests that the number of cores is a function which grows slowly on the input size n since there is a high processor cost. Let $\mathcal{P}(n)$ denote this function. Natural candidates for $\mathcal{P}(n)$ are $\Theta(\log n)$ and $\Theta(n^\alpha)$ for $\alpha < 1$, though there are other possibilities. Over the next subsections we shall consider various candidates for $\mathcal{P}(n)$.

3.2 Natural Constraints

The ability to index memory using a computer word as an address in a program's virtual memory suggests that the size of the word is $w = \Omega(\log M)$, where M is the memory size, though this does not necessarily need to be the case[1]. Memory itself is usually a polynomial function of the input size, i.e., $M = \Theta(n^k)$ for some $k \geq 1$, with $k = 1$ being a common value. Substituting $M = \Theta(n^k)$ in $w = \Omega(\log M)$ gives $w = \Omega(\log n)$. This is assumed in the word-RAM model, in order for algorithms to be able to refer to any input element. A common assumption in word-RAM papers is actually $w \approx \log n$, which enables constant-time lookup-table implementations of some functions on words while keeping table sizes sublinear (see, e.g., [23]), and restricts the size of pointers in succinct data structures that could otherwise increase their space usage (see, e.g., [9]).

Hence, the word size which in the early days of computing was treated as a constant, namely 4 or 8 bits, became better understood as in fact proportional to the logarithm of the input size, that is $\Theta(\log n)$. Similarly, in modern multi-core computers, the number of processors has remained

[1] In practice there have been architectures in which the memory size was strictly greater than 2^w. Currently in the Intel architecture the size w places a limit on the largest addressable space but this has not always been the case (e.g. the 8088 processor).

relatively bounded (in contrast to commercial PRAMs or GPUs which support anywhere from hundreds to hundreds of thousands of processors and still growing). This relatively slow growth (at least as compared to most other usually exponential growing performance hardware indices) on the number of processors can thus be best modeled as $\log n$ in similar fashion to the word size.

3.3 Write Conflicts

Consider a multi-threaded server application receiving requests from several clients simultaneously. Assume that these requests are served by parallel threads running on p processors that share the system's memory. Such an application is likely to have several portions of the computation accessing shared data such as database tables, buffers, and other shared data structures. Write accesses to shared data involve synchronization to avoid race conditions, usually implemented by synchronization primitives such as barriers and locks. In general, regardless of how synchronization is implemented, a simultaneous memory access to the same memory cell involves an overhead, either due to serialization, or data invalidation. Let us call a simultaneous access by a pair of threads a *collision*.

We are interested in analyzing the influence of the number of processors on the number of collisions during a period of computation. The uncertainty added by the timing of client requests suggests that write access to shared memory can be modeled as a random process with a certain probability of collision. A crude, but reasonable first order approximation is to consider memory access to shared data as uniformly random with p processors contending for access to memory.

We investigate the expected number of collisions for p threads accessing m memory cells, uniformly at random at each timestep of a period of service time. Clearly, the smaller the number of processors the lower the probability of collision. The question is for what value of p as a function of m does this probability become negligible. Note that in general the size of the memory is usually modeled as a growing function of a program's input size, with $m = O(n^k)$ being a common assumption. Thus it is reasonable to analyze the number of collisions as m grows.

This reduces to a balls-and-bins scenario (see, e.g., [16]). Let us first consider the total number of overall collisions in one step. Let C be a random variable denoting the number of collisions in one step. The probability that two memory accesses are to the same cell is $1/m$. Since the are are $\binom{p}{2}$ pairs of memory accesses, the expected number of collisions in one step is $E[C] = \frac{p(p-1)}{2m}$. As m grows this expression tends to 0 if $p < \sqrt{m}$, tends to infinity if $p > \sqrt{m}$, and to 1/2 for $p = \sqrt{m}$.

Now we consider an alternative expression for memory access conflicts, namely the number of cells involved in collisions at each step. Thus, if three or more accesses are to the same cell, the event counts as one collision. Let X be a random variable denoting the number of memory cells which suffer a collision when there are p simultaneous memory accesses. The probability of a memory cell not being accessed is $(1-1/m)^p$, and thus the expected number of accessed cells is $m - m(1-1/m)^p$. Then, the expected value of X is $E[X] = p - m + m\,(1-1/m)^p$. Assume that $p = m^\alpha$ with $\alpha \leq 1$. The expression above is then $E[X] \approx m^\alpha - m + me^{-m^{\alpha-1}}$. Using the Taylor expansion of $e^{-m^{\alpha-1}}$ we obtain $E[X] \approx \frac{m^{2(\alpha-1)}}{2}$.

Again, when m tends to infinity, the above tends to 0, 1/2, or diverges if α is less, equal, or greater than 1/2, and thus the threshold again is for $p = \sqrt{m}$. Clearly the smaller p is, the fewer the expected the collisions.

Case 1. If $p = m$, then $E[X] = m/e$, and $E[C] = (m-1)/2$. Thus in each step about 37% of memory cells have more than one processor trying to access them and about half of the accesses result in collisions.
Case 2. If $p = \sqrt{m}$ then on average there is a collision every two steps of an execution.
Case 3. If $p < \sqrt{m}$ the number of collisions goes to zero as m grows.

Suppose that every instruction takes unit time if there is no collision and $s \geq 1$ units of time otherwise. The expected number of collisions per processor per step is $\frac{(p-1)}{2m}$ and thus the expected slowdown in performance due to collisions is $\frac{s(p-1)}{2m}$, which is negligible for $p = o(m/s)$.

3.4 Processor Communication Network

Traditionally, parallel computers use either shared memory or a processor communication network (or both) to exchange information between the various processing units. The advantage of shared memory is that no additional hardware is required for it; the disadvantages are issues of synchronization and memory contention. Hence a widely explored alternative is the use of an ad-hoc processor communication network connecting the processors. In general, from the perspective of performance a full communication network is the preferable network architecture. However when the number of processors is assumed to be very large this is unfeasible. For example for the case of $\Theta(n)$-processors of many commercial PRAM implementations the number of interconnects required would have been $\Theta(n^2)$ which is prohibitive. Thus there was extensive study of alternative network topologies which reduced the complexity of the network while attempting to minimize the penalty in performance derived from the smaller network. Among the most successful such architectures we have the hypercube, the butterfly and the tori (see, e.g., [21]).

We observe now that full processor communication network becomes a realistic possibility if the number of processors is $O(\log n)$ or even possibly $O(n^{\alpha})$ for some $\alpha \ll 1/2$. For example for a modest (by present standards) input size of $100,000,000$ even $n^{1/2}$ processors would require and impossible number of interconnects on the full graph. A complete network of $O(\log n)$ processors on the other hand would require 300 interconnects which are well within the realm of current architectures.

3.5 Buffer overflow

Aside from issues of network topology, in practice it is natural to assume that each processor in a communication network can handle at most a small constant number of messages at once. If more than a constant number of processors send messages to a single processor, said messages would queue up at the receiving end for further processing. In this section we consider a natural communication model in which in each instruction cycle a processor may send a message to at most one other processor. In practice depending on the specific application the probability of collision may range anywhere from zero for the execution of independent threads to one for, say, a master processor serializing requests to some shared lock. As a compromise we model again this process as if the processors chose their destination uniformly at random. Let p be the number of processors; then the maximum number of collisions observed at the most loaded buffer is $O(\log p)$ with high probability [16]. If $p = \Theta(n)$ then buffer handling can introduce delays of 20–100 instruction cycles. In this case even $p = \Theta(n^{\alpha})$ for $0 \leq \alpha < 1$ might prove too costly. In contrast if we assume $p = \Theta(\log n)$ the most congested buffer would contain $O(\log \log n)$ elements which for all practical purposes is at most 6.

3.6 Divide-and-Conquer Algorithms

Divide-and-Conquer algorithms are naturally suited for parallelization. Instances at the same level of the recursion tree are independent and can be scheduled to be executed in parallel. This is especially well suited for multi-threaded systems, as each recursive calls can simply be handled by a separate thread. This strategy requires no parallelization of the divide and combine phases of the recursion, which can be executed by each thread just as in the sequential algorithm. It has been shown that this easy parallelization yields optimal speedups for a large class of divide-and-conquer algorithms [14], but only for a bounded number of processors. Thus, in a system with a logarithmic or sublinear number of processors, obtaining the maximum possible speedup for this class of algorithms is simple and can be realized with a general strategy that is independent of the algorithm itself.

Consider a divide-and-conquer algorithm whose time complexity can be written as $T(n) = aT(n/b) + f(n)$. The master theorem [13] yields the time bounds for a sequential execution of such an algorithm. A parallel version of this theorem can be obtained by analyzing the parallel time $T_p(n)$ of an execution in which recursive calls are executed in parallel and scheduled with the scheduler in [14] or work-stealing [11] with a bounded number of processors [14]:

$$T_p(n) = \begin{cases} O(T(n)/p), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \text{ and } p = O(n^{\epsilon}) & \text{(Case 1)} \\ O(T(n)/p), & \text{if } f(n) = \Theta(n^{\log_b a}) \text{ and } p = O(\log n) & \text{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 & \text{(Case 3)} \end{cases} \quad (1)$$

Optimal speedups are achieved in Cases 1 and 2 only for $p = O(n^{\epsilon})$ for $\epsilon > 0$, and $p = O(\log n)$, respectively. In Case 3, the time is dominated by the sequential divide and conquer time $f(n)$ at the top of the recursion [14].

We note that it is possible to obtain optimal speedups with larger numbers of processors for many divide-and-conquer algorithms. However, this invariably requires parallelizing the divide and combine phases of the algorithm, as otherwise the sequential time $f(n)$ of the divide and combine phases dominates the parallel time. In fact, if an optimal parallel algorithm for the divide and combination phases is known, then all cases above yield optimal speedup, and the bounds of the processors can be relaxed. Then the parallel time in Case 3 becomes $T_p(n) = \Theta(f(n)/p)$ [14]. Now Case 1 requires $p = O\left(\frac{n^{\log_b a}}{\log n}\right)$, Case 2 requires $p = O\left(n^{\log_b a}\right)$, while Case 3 requires $p = O(f(n)/\log n)$.

The result for a small number of processors shown in Equation (1) shows that for a system with a small number of processors the implementation of parallel divide-and-conquer algorithms that achieve the full speedup offered by the architecture is simple and can be implemented without the unnecessary complexity of implementing specific parallel algorithms for the divide and combine phases of the algorithms.

When considering cache performance of divide-and-conquer algorithms, a bounded number of processors can also be advantageous. Blelloch *et al.* [6] show that the class of *hierarchical* divide-and-conquer algorithms —algorithms in which the divide and combine phases can also be implemented as divide-and-conquer algorithms— can be parallelized to obtain optimal speedups and good cache performance when scheduled with a Controlled-PDF scheduler. While a Brent's Lemma type of implementation of some of the algorithms in [6] can achieve optimal speedups for a large number of processors (e.g., matrix addition and cache oblivious matrix multiplication algorithms can both be sped up optimally up to n^2 processors) [6], the optimal speedup and cache performance bounds under the Controlled-PDF scheduler is only achieved for a much smaller number of processors, bounded by the ratio between shared and private cache sizes, and even smaller in some cases, as we shall see in the next section.

3.7 Cache imposed bounds

Cache contention is a key factor in the efficiency of multi-core systems. Various multi-core cache models have been studied, with a focus on algorithms and schedulers with provable cache performance. Many of the results involving shared and private caches performance require bounds on the number of processors related to the relative size of the private and shared caches.

The Parallel External Memory (PEM) model [3] models p processors, each with a private cache of size M, partitioned in blocks of size B. A sorting algorithm given in this model is asymptotically optimal for the I/O bounds for at most $p \leq n/B^2$ processors, and it is actually proven that $p \leq n/(B \log B)$ is an upper bound for optimal optimal processor utilization for any sorting algorithm in the PEM model [3]. This algorithm is used in further results in the model for graph and geometry problems [4, 1, 2]. Thus the assumption that $p \leq n/B^2$ is carried on to these results as well, some of which actually require $p \leq n/(B \log n)$ and even $p \leq \frac{n}{B^2 \log B \log^{(t)} n}$, where $\log^{(t)} n$ denotes the composition of t log functions, and t is a constant.

Shared cache performance is studied in [7], which compares the number of cache misses of a multi-threaded computation running on a system with p processors and shared cache of size C_2 to those of a sequential computation with a private cache of size C_1. It is shown that under the PDF-scheduler [8], the parallel number of misses is at most the sequential one if $C_p \geq C_1 + pd$, where d is the critical path of the computation. This implies that $p \leq (C_p - C_1)/d$, which is less than n (as otherwise all the input would fit in the cache) and is usually sublinear, as d is rarely constant and is $\Omega(\log n)$ for many algorithms. Thus, for many algorithms the bound on the parallel misses hold for $p = O(n/\log n)$.

As mentioned in Section 3.6, Blelloch *et al.* [6] study hierarchical divide-and-conquer algorithms in a multi-core cache model of p processors with private L_1 caches of size C_1 and a shared L_2 cache of size C_2. An assumption of the model is that $p \leq \frac{C_2}{C_1} \ll n$, since the input size is assumed not to fit in L_2. It is shown that under a Controlled-PDF scheduler, parallel implementations achieve optimal speedup and cache complexity within constant factors of the sequential cache complexity for a class of hierarchical divide-and-conquer algorithms. Optimality for some algorithms, such as Strassen's matrix multiplication and associative matrix inversion even require $p \leq (C_2/C_1)^{\frac{1}{1+\epsilon}}$ [6].

Cache efficient dynamic programming algorithms have been designed in this multi-core model with the same $p \leq \frac{C_2}{C_1}$ assumption [12], as well as in a shared cache model with $p \leq C_2/B$, where B is the block size. Thus although the time complexity of parallel dynamic programming allows a large number of processors for optimal speedups (e.g., $T_p = O(n^3/p + n)$ for Gaussian elimination paradigm problems, which is optimal for $p \leq n^2$), the efficiency in cache performance restricts the level of parallelism.

Observe that presently the ratio between L_2 shared cache and private L_1 cache is in the order of 4 to 100 depending on the specific processor architecture.

3.8 The class $E(p(n))$

The class NC can be defined as the class of problems which can be solved in polylog time using polynomially many processors. It is believed that $NC \neq P$ and hence that there are known problems which do not admit a solution in time $O(\log^k n)$, for some $k \geq 1$. In our case we are interested in the study of problems which can be sped up using $O(\log n)$ or $O(n^\alpha)$ processors for $\alpha < 1$. Kruskal et al. [20] introduced the classes ENC and EP which encode the classes of problems that allow optimal speed up (up to constant factors) using polynomially many processors on a CRCW PRAM. The class ENC has polylogarithmic running time, while the class EP has polynomial running time. They also define the related classes SNC, ANC, SP, and AP, which are analogous to ENC and EP in terms of the required running times but allow for some inefficiency. In general, one could introduce the class $\mathcal{C}(p(n), S(n))$ as the class of problems that allow a speedup of $S(n)$ with $p(n)$ processors. Thus, following the notation in [20] we define the class $E(p(n)) = \mathcal{C}(p(n), p(n))$, which is the class of problems that can be solved using $O(p(n))$ processors in time $O(T(n)/p(n))$ where $T(n)$ is the running time of the best sequential solution to the problem. In this work we are particularly interested in the classes $E(\log n)$ and $E(n^\alpha)$ for $\alpha < 1$ [2].

The class ENC is a sharpening of the well known class NC. Recall that the class NC requires maximal speedup down to polylogarithmic time even at the cost of a polynomial amount of inefficiency (i.e., the ratio between parallel and sequential work). In contrast ENC requires the same speedup but bounds the inefficiency to a constant factor.

The class $E(\log n)$ bounds the inefficiency to a constant which implies a speed up of $\Theta(\log n)$ on the sequential solution to the problem. Observe that by Brent's Lemma $ENC \subset EP \subset E(\log n)$:

Theorem 1. $ENC \subset EP \subset E(\log n)$

Proof. The inclusion $ENC \subset EP$ is stated in [20]. We now show that $EP \subset E(\log n)$. Let $\Pi \in ENC$ be a problem with sequential running time $t(n)$. Let A be an algorithm that solves Π in time $O(t(n)^\epsilon)$ with p processors, where $\epsilon < 1$ [20]. Since A is work-optimal the total work done by A is $O(t(n))$. Then, by Brent's Lemma [10], we can simulate A in $T'_p(n) = O(t(n)/p' + t(n)^\epsilon)$ time with $p' \leq p$ processors. The simulation achieves optimal speedup for any $p' = O(t(n)^{1-\epsilon})$. Since for $E(\log n)$ we have $p' = O(\log n)$, which is $O(t(n)^{1-\epsilon})$ for any $t(n) = \Omega((\log n)^{1/(1-\epsilon)})$, then $\Pi \in E(\log n)$. □

The reverse is not the case, i.e., $E(\log n) \neq ENC$, and $EP \neq ENC$ unless $P = NC$ since there are known P-complete problems which allow optimal speedup using a polynomial number of processors [17], and thus they are in EP (and hence in $E(\log n)$). If any such problem is in ENC, this would imply $P = NC$. We conjecture that $E(\log n) \neq EP$ as well.

Similarly $E(n^\alpha)$ bounds the inefficiency to a constant which implies a speed up of $O(n^\alpha)$ on the sequential solution to the problem. Again we have $ENC \subset EP \subset E(n^\alpha)$, while $E(n^\alpha) \neq ENC$ for any $\alpha < 1$ for the same reason described above. Again, we conjecture that $E(n^\alpha) \neq EP$.

Theorem 2. $ENC \subset EP \subset E(n^\alpha)$

Proof. The proof is analogous to the proof of Theorem 1. The simulation of an algorithm for a problem $\Pi \in EP$ using Brent's Lemma is optimal up to $p' = O(t(n)^{1-\epsilon})$ processors, which is $\Omega(n^\alpha)$ (with $\alpha < 1$) for any $t(n) = \Omega(n^{1/(1-\epsilon)})$, which holds in particular for any $t(n) = \Omega(n)$. □

This gives a theoretical separation between the problems that can be speed up optimally using polynomially many processors and those that can be speed up using a logarithmic number of processors. We conjecture that the separation holds as well for $O(n^\alpha)$ processors for any $\alpha < 1$.

[2] For consistency in the class comparisons, we assume a CRCW PRAM as in [20], though these classes can be defined for other PRAM types (EREW,CREW), as well as for asynchronous models (such as multi-cores).

3.9 Optimal Time Simulation of Turing Machines by $O(\log n)$ Processors

We show that any computation on a Turing machine that takes time $T(n) \geq n \log n$ can be carried out in parallel by a multi-core system with $p = \lg n$ processors in time $T_p(n) = O(T(n)/(\lg n + \lg\lg n))$. There are known simulation results for Turing Machines by a sequential RAM [19] as well as by a PRAM [15]. In the latter it is shown that a deterministic machine running in $T(n)$ time can be simulated by a PRAM in time $O(\sqrt{T(n)})$ using an exponential number of processors and memory addressing on words of size $O(\sqrt{T(n)})$. We adapt this simulation to a more realistic logarithmic number of processors and word size.

Outline. Let M be a single-tape deterministic Turing Machine[3]. The idea of the simulation is to treat contiguous blocks of $b = b(n)$ bits of M's tape as a word in RAM. By precomputing M's resulting configuration after b steps when starting with each possible block, we can then simulate b steps of M at a time by successively looking up the next configuration of M from the precomputed table. Let $g(n)$ denote the precomputation time. If each access to the precomputed table takes constant time, then the total time of the simulation is $T_p(n) = \frac{T(n)}{b(n)} + g(n)$.

Precomputation phase. Since in b steps M can only alter the contents of b cells, for a given position within the tape we need only to consider the content of the b cells to the left and b cells to the right of the current position in order to compute the resulting configuration after b steps. A block configuration of M is a tuple (s, B), where s is a state, and B is a $(2b-1)$-bit string representing the contents of a segment of M's tape around some position of the head. For each possible block configuration c, we store in $A[c]$ the resulting configuration when running M starting from c (i.e., the new state and block contents), plus information about how many positions the head moved, and in which direction. The latter is necessary to know where the new block should be centered in M's tape. A block configuration c uses $|c| = 2b - 1 + d = O(b)$ bits, where d is the constant number of bits required to indicate a state of M. Let k be a constant such that $|c| \leq kb$. Then there are at most 2^{kb} starting block configurations. Since the resulting configurations starting from all possible configurations can be computed independently in parallel and each computation takes $O(b)$ time by direct simulation of M, the total precomputation time is $g(n) = O\left(2^{kb} b/p\right)$ using p processors. Note that the precomputation requires M's specification but is independent of a particular input.

Simulation phase. Suppose the configuration table A has already been computed and it is stored in the RAM of the multi-core machine. If the length of each configuration is smaller than the machine word's length, then A can be indexed by configuration and each entry can be accessed in constant time. For this sake we set $b = \frac{\lg n + \lg\lg n}{k}$, and thus $|c| \leq \lg n + \lg\lg n = \Theta(\log n)$. Therefore A can be stored as an array of configurations, indexed by configurations. Given M and an input x, and starting with the initial configuration c_0, the multi-core simulates M (using one processor) by applying $c_{i+1} = A[c_i]$, and updating the contents of M's tape at each step, until c_{i+1} contains a final state. Since at each step A can be accessed in constant time and the relevant part of M's tape can be updated in constant time, the simulation takes $O(T(n)/(\lg n + \lg\lg n))$ time and the precomputation takes $O\left(\frac{2^{\lg n + \lg\lg n}(\lg n + \lg\lg n)}{p}\right)$ time, which is $O(n \lg n)$ for $p = \lg n$. Thus the total time is

$$T_p(n) = O\left(\frac{T(n)}{\lg n + \lg\lg n} + n \lg n\right)$$

[3] It is straightforward to extend the simulation to a k-tape Turing Machine.

Faster recursive precomputation. The approach described above requires $T(n) = \Omega(n \lg^2 n)$ to be optimal. However, we can relax this requirement by speeding up the precomputation phase. This approach, whose details are described in Appendix A, yields a simulation time of $T_p(n) = O\left(\frac{T(n)}{\lg n + \lg\lg n} + \frac{n \lg n}{p} + \lg\lg p\right)$, which for $p = \lg n$ and $T(n) \geq n(\lg n + \lg\lg n)$ is $O\left(\frac{T(n)}{\lg n + \lg\lg n}\right)$.

Larger number of processors. A simulation like the one described above would not be optimal if $p = \omega(\lg n)$. If the length of the blocks is kept at $b = O(\lg n + \lg\lg n)$, then the total time is $T_p(n) = \Omega\left(\frac{T(n)}{\lg n + \lg\lg n} + \frac{n}{p} + \lg\lg p\right)$, which for $p = \omega(\lg n)$ can never be $O(T(n)/p)$. A longer block length $b = \omega(\lg n)$ could reduce the simulation phase time, but would require an infeasible superpolynomial-size precomputed table. This, of course, does not preclude the existence of other approaches that could result in optimal simulation time with a larger number of processors.

3.10 Amdahl's Law

Consider a program whose execution has a serial part that cannot be parallelized (unless $P = NC$) represented by $S(n)$ and a fully parallelizable part denoted by $P(n)$ then the parallel time with p processors is: $T_p(n) = S(n) + P(n)/p$ and the speedup is represented by

$$\frac{T_1(n)}{T_p(n)} = \frac{S(n) + P(n)}{S(n) + P(n)/p}.$$

Observe now that for $p = \Theta(n)$ we get that the parallel program is noticeably faster only if $S(n) = O(P(n)/n)$. For $p = \Theta(n^\alpha)$ we get that the parallel program is noticeably faster only if $S(n) = O(P(n)/n^\alpha)$. Lastly, for $p = \Theta(\log n)$ we get that the parallel program is noticeable faster if $S(n) = O(P(n)/\log n)$. Observe that most practical algorithms on large data sets run in time $O(n \log n)$ or less, with the sequential part often corresponding to I/O operations, i.e., reading the input. This means that the likeliest value for which one can obtain optimal speedup corresponds to $P(n)/S(n)$ which is often (though not always) $\log n$.

4 Conclusions

We presented a list of theoretical arguments and practical evidence as to the existence of a qualitative difference between the classes of problems that can be sped up with a sublinear number of processors and those that can be sped up with polynomially many processors.

We also show that in various specific instances even though there are optimal algorithms for either case, it is conceptually and practically much simpler to design an algorithm for a sublinear number of processors. The benefits of a low processor count extend to issues of processor communication, buffering, memory access, and cache bounds.

We introduced classes that describe the problems that allow for optimal speed up, up to a constant factors, for logarithmic and sublinear number of processors and show that they contain a strictly larger class of problems that the PRAM equivalents introduced by Kruskal, Rudolph, and Snir in 1990 [20], unless $NC = P$.

The discontinuities identified in behaviour and performance of parallel systems for logarithmic and sublinear number of processors make these particular processor count functions theoretically interesting, practically relevant, and worth of further exploration.

References

1. Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. Geometric algorithms for private-cache chip multiprocessors - (extended abstract). In Mark de Berg and Ulrich Meyer, editors, *ESA (2)*, volume 6347 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 2010.
2. Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. I/o-optimal distribution sweeping on private-cache chip multiprocessors. In *IPDPS*, pages 1114–1123. IEEE, 2011.
3. Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In Meyer auf der Heide and Shavit [22], pages 197–206.
4. Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *IPDPS*, pages 1–11. IEEE, 2010.
5. Michael A. Bender and Cynthia A. Phillips. Scheduling dags on asynchronous processors. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 35–45, New York, NY, USA, 2007. ACM.
6. Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandrana, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the 2008 ACM-SIAM Symposium on Discrete Algorithms*, January 2008.
7. Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.
8. Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46:281–321, March 1999.
9. Prosenjit Bose, Eric Y. Chen, Meng He, Anil Maheshwari, and Pat Morin. Succinct geometric indexes supporting point location queries. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 635–644, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.
10. Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.
11. F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.
12. Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In Meyer auf der Heide and Shavit [22], pages 207–216.
13. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
14. Reza Dorrigiv, Alejandro López-Ortiz, and Alejandro Salinger. Optimal speedup on a low-degree multi-core parallel architecture (lopram). In Meyer auf der Heide and Shavit [22], pages 185–187.
15. Patrick W. Dymond and Martin Tompa. Speedups of deterministic machines by synchronous parallel machines. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 336–343, New York, NY, USA, 1983. ACM.
16. William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.
17. Akihiro Fujiwara, Michiko Inoue, and Toshimitsu Masuzawa. Parallelizability of some p-complete problems. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 116–122, London, UK, UK, 2000. Springer-Verlag.
18. Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Inc., New York, NY, USA, 1995.
19. John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. On time versus space and related problems. In *FOCS*, pages 57–64. IEEE, 1975.
20. Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.*, 71(1):95–132, March 1990.
21. F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
22. Friedhelm Meyer auf der Heide and Nir Shavit, editors. *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*. ACM, 2008.
23. J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

A Faster Recursive Precomputation Phase of Turing Machine Simulation.

The idea is to recursively apply the simulation on the computation of each entry of A. Let g_i and b_i denote the precomputation time and block length of the i-th level simulation, respectively. Thus $g_m = g(n)$ is the total precomputation time and $b_m = b(n)$ is the block length of the final simulation as described above. Since the computation of each entry of A can now be sped up by b_{m-1}, we have

$$g_m = \frac{2^{kb_m}}{p}\frac{b_m}{b_{m-1}} + g_{m-1},$$

where k is a constant such that for all i, a configuration in level i has size at most kb_i. We then set $b_{m-i} = \frac{\lg n + \lg\lg n}{k2^i}$ for all $0 \le i \le m = \lg\left(\frac{\lg n+\lg\lg n}{k}\right)$. Then, $b_{m-i}/b_{m-i-1} = 2$, which is the length of the critical path at each recursive level. Then,

$$g_{m-i} = \max\left\{\frac{2^{kb_i+1}}{p}, 2\right\} + g_{m-i-1} = \max\left\{\frac{2(n\lg n)^{\frac{1}{2^i}}}{p}, 2\right\} + g_{m-i-1}$$

Note that $2(n\lg n)^{\frac{1}{2^i}}/p \le 2$ when $i \ge \lg\left(\frac{\lg n+\lg\lg n}{\lg p}\right)$. Let $i^\star = \lg\left(\frac{\lg n \lg\lg n}{\lg p}\right)$. Since $g_0 = 0$,

$$\begin{aligned}
g_m &= \sum_{i=0}^{m-1} g_{m-i} - g_{m-i-1} \\
&= \frac{2}{p}\sum_{i=0}^{i^\star}(n\lg n)^{\frac{1}{2^i}} + \sum_{i=i^\star+1}^{m-1} 2 \\
&\le \frac{2}{p}(n\lg n + i^\star\sqrt{n\lg n}) + 2(\lg\lg p - \lg k - 1) \\
&\le \frac{4n\lg n}{p} + 2\lg\lg p
\end{aligned}$$

Therefore, the total simulation time is now $T_p(n) = O\left(\frac{T(n)}{\lg n+\lg\lg n} + \frac{n\lg n}{p} + \lg\lg p\right)$, which for $p = \lg n$ and $T(n) \ge n(\lg n + \lg\lg n)$ is $O\left(\frac{T(n)}{\lg n+\lg\lg n}\right)$.