Snapshots from the frontiers of computing

Sudebkumar Prasant Pal[1]

Great strides in technological advancements over the last three decades have taken computers from very huge and slow machines of the 70s and 80s to the light and fast palm-tops of today. Memory sizes have fallen and their speeds have gone up tremendously. Better processor architectures have revolutionized human-computer interactions. Consequently, a personal computer (PC) today is not only a general purpose computing device for the professional, it also provides limited audio and video entertainment facilities. Light laptops, mobiles and other hand-held devices with wireless communication facilities have changed the way people conduct business. Networking technologies have provided international connectivity in addition to local area networks and local area wireless networks. The rapid growth of the world wide web and the rapid advancements in internet technologies has brought human beings closer; the internet has certainly made commerce, trading and business more easy, fast and cheaper across huge distances.

The computing revolution is remarkable. The micro-miniaturization in processor and memory chips and effective computer aided design technologies for VLSI are largely responsible for the current state-of-the-art in the design and fabrication of computing devices. These developments have made faster and cheaper computing facilities available on the desktop. With continuously improving computer architectures, existing methods of problem solving have been running faster than before. The same old algorithms for solving a system of linear equations and the same simplex method run faster on problems of larger sizes in today's machines. A relevant question here is whether there has been remarkable progress in the very methods of problem solving. Have there been major breakthroughs in paradigms of the design of algorithms for important classes of problems? Are there important problems of considerable magnitude that can not be solved efficiently using today's computers? What are the notions of efficiency in computing? Are there any fantastic newer computing methods coming up in the future? We will attempt to address some of these issues in this article.

1 Challenges in parallel computing

How much power does one have on his PC? Even with the best PCs, one may have just a single processor or at most a dual processor. So, for performing really big simulations, one has to connect to servers that are more powerful. Servers for scientific computing could typically provide modern day parallel processing capabilities. Such a 'supercomputer' should be able to really exploit parallelism in the algorithm for the scientific computing problem at hand, saving precious computing time.

1.1 Pragmatic parallel computing

Typically, parallel computers have fairly closely knit processors, coupled with huge banks of memory. Interconnection networks are used to parallelize data movements between multiple processing units. The computer scientist's real challenge here is to be able to map algorithms into the interconnection networks, thereby exploiting the inherent implicit parallelism (if any) in the algorithm. Extensive research has gone into this field of mapping algorithms to parallel computers. Another approach has been the utilization of multiple PCs for solving a big problem parallelly, using communication across a fast local area network. Here, the PCs are quite loosely coupled, and, might be doing independent local jobs; their idle times could very well be used to advance the parallel computing jobs they are cooperating to solve. Suitable software and hardware solutions have been developed for such use of PCs for doing substantial parallel processing.

[1]Department of Computer Science and Engineering, IIT Kharagpur, email:spp@cse.iitkgp.ernet.in
url- http://www.angelfire.com/or/sudebkumar

Analytical studies and practical experience reveal that merely increasing the number of processors does not always decrease the processing time. Processing time may not decrease continuously if the number of processors is raised indefinitely. In that sense, practical parallel computing is not scalable. One reason why this happens is that there are considerable overheads in communicating data between processors. Parallel computing is quite as much about communicating data between processors, as it is about partitioning computing load between processors. Considerable degree of communication is required between processors for solving non-trivial problems of large sizes.

1.2 Parallelisability of problems

The study of inherent parallelism in solutions for problems can help us estimate the degree of speedups we can achieve. It is true that n additions can be done in constant time if n processors can access the n pairs of summands parallelly and store the results simultaneously. This requires parallel access to the memory by multiple processors and synchronization of the processors. A favourite (unrealistic) model of parallel computing studied by computer scientists is that of a *shared memory* connected to an unlimited number of processors. These processors can be allocated to do elementary units of computation as required. The processors can simultaneously access memory locations in the shared memory. Such a computing model is very powerful indeed, giving full speedup for the problem of computing the maximum of n numbers in $c \log n$ time, where c is a constant independent of n. The number of processors required is just $bn/\log n$ (here b is a constant independent of n), making the processor-time product proportional to n. [2] Thus the total work done here is bounded by a constant multiple of the work required to do $n-1$ comparisons (for computing the maximum of n numbers) in a fully sequential process. This is *optimal speedup* with sublinear number of processors. It is also possible to perform the sorting of n numbers with full speedup using n processors in this shared memory model; the optimal parallel computation time of $O(\log n)$ is achieved[3]. This result of the mid-eighties is due to Richard Cole.

Many fundamental computational problems in *graph theory* and *computational geometry* have been shown to have fast parallel algorithms with optimal or near-optimal speedups. For instance, the *convex hull* of n points in the plane can be computed in $O(\log n)$ time using $O(n)$ processors. This result was worked out by Michael Goodrich in the eighties, following Cole's sorting result. The speedup here is optimal because the total work done by the processors is $O(n \log n)$, which is also the best running time for the same problem in usual sequential computation. Note that in all problems we consider, n measures the size of the input, or more generally, n represents a quantity that is proportional to the actual input size - the number of elementary data elements in the input to an algorithm for the problem. It could be the number of elements to be sorted, the number of points in the plane, or the number of edges and vertices in a graph or network. When we say that a sequential algorithm runs in polynomial time, we mean that it runs in time proportional to a polynomial in the length n of its input.

1.3 The logarithmic obsession

Note that we have been very particular about running our parallel algorithms in $O(\log n)$ time, where n is the input size. This is natural because the logarithm function is an exponential order smaller than the linear function, say cn, where c is a constant. We would be quite happy to have the parallel running time of an algorithm bounded by low degree polynomials over $\log n$, where n is the input size of the problem instance being solved. We may not have an $O(\log n)$ time parallel algorithm for a certain problem; in that case an

[2] If $p(n)$ processors require parallel time of $t(n)$ to carry out a certain parallel computation, then the total work done is $p(n)t(n)$. A simulation with one processor would require at least $p(n)t(n)$ time.

[3] The Big-Oh or 'O' notation is used to represent upper bounds. A function $g(n)$ (which may be the maximum running time of an algorithm over all inputs of length n), is written as $O(f(n))$ if for all $n > n_0$ (where n_0 is some integer), there is a constant $C > 0$ such that $g(n) \leq Cf(n)$.

$O(\log^k n)$ (k is a constant) time parallel algorithm would be considered quite a fast algorithm as well. Such execution time is popularly called *polylogarithmic*. We have to however restrict the number of processors to a polynomial in n. A problem is said to have an *efficient parallel algorithm*, if there is a parallel algorithm in the shared memory model that solves the problem in polylogarithmic time with a polynomial number of processors. There exist many problems solvable sequentially in polynomial time which are not known to have efficient parallel algorithms.

The computation of Voronoi diagrams of a set of points, the computation of intersections between a set of line segments, and the computation of triangulations of polygons and points sets are problems that have efficient parallel algorithms. The *minimum link path* problem is that of computing the path with the minimum number of bends connecting two points inside a polygon. This problem was shown to have an efficient parallel algorithm by Chandru, Ghosh, Maheshwari, Rajan and Saluja. The minimum link path problem is relevant in robotics and motion planning where we wish the robot to take a minimum number of turns on the way to its destination.

A very important problem in graph theory is that of determining whether there is a *path* from one vertex s to another vertex t. This is called the *graph reachability problem*. A *path* is a sequence of edges where the first edge connects the vertex s and the last edge connects the vertex t. Any two consecutive edges in the *path* share a vertex. The graph reachability problem has efficient parallel algorithms. Problems from scientific computing like inverting a matrix, computing determinants and ranks of matrices also have efficient parallel algorithms. Harder problems like certain kinds of simple dynamical programming problems, maximum flow in planar networks and linear programming with a fixed number of variables also have efficient parallel algorithms.

The significance of the fact that many useful problems have efficient parallel algorithms is that practical parallel architectures are likely to be able to exploit the inherent parallelism in these problems. It is likely that parallel computers would be able to actually solve some such problems in time comparable to the theoretically established bounds possible in the shared memory model discussed above. Specific parallel architectures can also be used to solve specific problems efficiently, in polylogarithmic time.

1.4 Problems that are hardly parallelizable

We will particularly concentrate on problems that give only 'yes' or 'no' as output. These are called *decision problems* and are conveniently used for defining classes of problems. As we mentioned above, certain problems having polynomial sequential running times are not known to have efficient parallel algorithms. Indeed, it can be shown that if some special problems having (sequential) polynomial time algorithms also had efficient parallel algorithms, then all decision problems in the class P [4] would also have efficient parallel algorithms. Such problems are called *complete for the class P* or simply *P-complete*. One such problem (MFODD defined below) is a simpler decision version of the *maximum network flow* problem. A *network* is given where each *edge* in the network has some capacity of permissible flow. We wish to pump the maximum possible flow F, into an arbitrary *node* in the network s, send the flow through edges of the network obeying flow capacity constraints in the edges, and take the flow F out of another arbitrary node t. Computing the maximum flow F given the network and the nodes s and t, is a problem that is not likely to have an efficient parallel algorithm. MFODD is the problem of deciding whether the maximum flow in a network from s to t is an odd number. We know that MFMOD is P-complete. If we had an efficient parallel algorithm for computing the maximum network flow in a network, then MFODD would also be solvable in polylogarithmic time with polynomial number of processors. MFODD being P-complete and possesing an efficient algorithm would force all problems in P to

[4] The class P is the set of decision problems such that any problem in it has an algorithm with running time bounded by a polynomial in the size of the input given to the algorithm. These are supposed to be problems that require reasonable or low time resource and, are generally known as tractable problems. We say that problems in the class P have polynomial running times.

be in NC [5], thereby making NC=P, which many feel is unlikely.

2 Playing dice with computers

Some practically useful speedups may sometimes be achieved for problems whose known algorithmic solutions are expensive in terms of running time. Many of these problems are very hard and may require prohibitively large number of processors for achieving fast running times on parallel machines. If you use a limited number of processors, the running times would not be appreciably smaller than what you would get with a sequential computer. One such hard problem is that of testing whether an n-digit number N is a prime. One would be tempted to divide N with numbers less than N to test the primality of N. Even if you divide with numbers that are less than the square-root of N, the number of divisions required is an exponential in the size n of the input number N. Clearly, such an algorithm does not force primality testing to belong to the class P. To date, it is not known whether primality testing belongs to the class P.

2.1 Guess and verify

Inspite of the difficult scenario about primality testing, we do find solace in the fact that given an n-digit number N and a *short proof*, we can process N and the short proof in polynomial time to verify that N is indeed a prime. The short proof would usually be a string of numbers provided by a hypothetical *agent* called the *prover*. It is essentially intended to be a certificate that the prover provides to a *verifier*, trying to convince the verifier that N is indeed a prime due to the proof encoded in the short proof certificate. The total length in number of bits in the short proof would be a polynomial in n. If the prover who provides the short proof falters or lies, then the verification would fail and we would not know whether N is a prime. Otherwise, assuming that the prover is sincere and capable, and N is indeed a prime, the prover would really give a correct short proof that we can verify in time proportional to a polynomial in n. Such a prover-verifier protocol is possible for a certain class of problems, widely known as the class NP [6]. Here the protocol essentially consists of a suitable verifier algorithm that checks the short proof against the input. Primality testing and testing a number for compositeness are problems in the class NP. We should be thankful that these problems are checkable in polynomial time in the above sense. There are other problems which are provably unsolvable in polynomial time; these problems do not belong to the class P. Although we know that there are problems which do not belong to P (these are problems that require provably exponential time to solve), we are not aware of any problem that belongs to NP but not P. We also do not know yet whether primality and compositeness (or for that matter, problems in NP) belong to the class P.

It is very easy to state the prover-verifier protocol for the compositeness problem; given a divisor D of a composite number N, verifying that D is indeed a divisor requires a simple division of N by D. If the remainder is zero, N is indeed checked to be composite. If the prover has given a D which does not divide N, or N is a prime, then the checking for divisibility with D would never succeed. The prover-verifier protocol for primality testing is due to Pratt; being more complex, we do not discuss it here. Remember that all proof checking that we do here must take time proportional to the size of the input; the size of the short proof is also polynomial in n, the size of the input.

All what we have discussed so far does not carry us towards any practical method for primality testing. Surely, very expensive methods of brute force can certainly solve primality and compositeness problems, after spending time proportional to an exponential function in the number of digits of the number to be tested.

[5] NC is the class of all problems in P that have an efficient parallel algorithm, one that runs in polylogarithmic time with polynomial number of processors.

[6] NP stands for *nondeterministic polynomial*; nondeterminism essentially captures the notion of the guess (short proof) provided by the prover. The verification is done in polynomial time.

For large numbers running into hundreds and thousands of digits, the time required to test primality or compositeness would be formidable and would run into years and decades, even using the fastest computers.

2.2 Toss a coin, do not guess

What about allowing a little bit of uncertainty or inexactness in problem solving? We have to compromise certainty because we still wish to run algorithms for solving tough problems in time proportional to a polynomial in the size of the input. Suppose you had an algorithm A (for compositeness) which takes a number N as input and runs in time proportional to a polynomial in the number of digits in N, finally giving a yes/no answer. Suppose that A certainly says 'no' if the number is a prime, and, says 'yes' with a finite (constant) probability if the number is composite. The algorithm uses coin tosses to choose paths of execution. At certain stages during the execution of the algorithm, a coin would be tossed to decide which branch (say one out of two or more) will be taken next. The algorithm A surely fails to tell the right answer for some runs on composite inputs; A may also make conflicting decisions on the same composite input in two different runs. Nevertheless, it does quite well because its probability of success for composite inputs is finite, however small. It is sometimes convenient to assume this success probability is not less than 50%; unless mentioned otherwise, we will assume that the success probability is indeed at least one-half. There is indeed such an algorithm for testing compositeness of numbers, proposed in 1980 by Rabin. Computational problems that admit such randomized algorithms are said to constitute the class RP (RP stands for *randomized polynomial time*). Rabin's algorithm therefore places the compositeness problem in the class RP. Also note that if A tells 'yes' on a certain input, we are sure that it is composite; if it says 'no' then we do not know for sure whether the input was composite or prime. (In the rest of this article, we will restrict ourselves to decision problems whose output answers are 'yes' or 'no', unless specifically mentioned otherwise.)

On the other hand, there is a polynomial time algorithm B that surely says 'no' for composite inputs, and says 'yes' for prime inputs with a finite probability. This algorithm is also inexact like A because B may fail to tell that a prime input given to it was indeed a prime. Such an algorithm was given in 1987 by Adleman and Huang. Like A, B too runs in time polynomial in the size of its input. This algorithm places the primality testing problem in the class RP. Again, note that as long as B says 'yes', we can be sure that the input was indeed a prime. Otherwise, we are unsure about the primality of the input.

2.3 Amplify success probabilities

The algorithms A and B are one-sided; A (B) correctly tells that its input is a prime (composite number) if the input is indeed so. Otherwise, A (B) has a finite probability of telling correctly that the input is indeed composite (a prime). The finite probabilities of failure of A (B) when the input is composite (a prime), can be made smaller and smaller by running A (B) a number of times. Indeed, the polynomial time algorithm A (B) (requiring $p_1(n)$ time) can be run a polynomial $p_2(n)$ number of times in order to reduce the error probability of failing to recognize and declare a composite (prime) input correctly. Fortunately, the error probability can be reduced to an inverse exponential, of about $2^{-p_2(n)}$. The polynomials and exponential functions here are functions of the input length n (number of digits) of the input N.

Now we have a practical method of testing primality/compositeness, as follows. Run A followed by B, repeatedly on the same input integer N. If at any stage A (B) says 'yes' and thereby declares N is a composite number (prime), we are done and we stop. If both A and B say 'no', we keep continuing because we know that A (B) has a finite probability of error when it says 'no'; a 'no' answer can come from each of A and B for prime as well as composite inputs. However, the probability that both A and B keep telling 'no' over a large number of independent runs is very small. (This is easy to see; without loss of generality, suppose N is composite (prime). Then, the probability that A (B) says 'no' is less than 50%. The probability that A (B) keeps telling 'no' over a number of independent trials diminishes. On the other hand B (A) can not tell 'yes' for a composite (prime) input.) If the number of stages is $p_2(n)$, this probability is proprtional to an inverse exponential in $p_2(n)$ as mentioned above. So, it is *expected* that either A or B will eventually tell 'yes', settling the primality of the input. We can not say at which stage this will happen but we *expect* that this will happen in practice at some stage. The *expected running time* until the correct decision is taken can be shown to be a polynomial in n. Computer scientists and programmers consider this to be a great success. What we have now is a mechanism to surely test primality (or equivalently compositeness) of any input integer N with running time expected to be a polynomial in the number of digits of N. Problems admitting such algorithms are said to form the class ZPP (ZPP stand for *polynomial time randomized algorithms with zero probability of error*).

2.4 Tossing a coin inside the computer

We assumed so far that we can toss a coin inside the computer. This means that we can generate sequences of *random bits*. For this we need some *perfect random source* which would indefinitely pour out zeros and ones with equal probability. Moreover, all sequences generated of any finite length n must be equally probable, with probability 2^{-n}. This means that every random bit is generated independently and its value does not depend on the history of bits generated earlier. It is not clear whether nature can provide such a random source. Computers surely can not provide such sources by themselves. They always run *deterministic algorithms* whose outcomes are totally predictable; given all the inputs (hidden or explicit) to any algorithm X, a master simulator algorithm can always predict the outcome of X, correctly. The problem we face is therefore of belling the cat; we must have some way of generating some sort of random behaviour in our deterministic machines, so that we can run algorithms like the one alluded to in Section 2.3.

We will make a compromise and use a (imperfect) *slightly random* source to generate bits where the ith bit is dependent on all the previous $i-1$ bits in some (limited and controlled) way. So, we relax the independence criterion but quantitatively limit the dependence of a random bit on previous random bits. In Section 2.3 we used random bits from a perfect random source to decide branchings in the execution of the randomized algorithms A and B. Now we have only the *slightly random* source as mentioned above. We must use random bits coming out of the slightly random source and show that we can still simulate the behaviours of algorithms A and B. Firstly, we must ensure that our new modified algorithm A' (B') for A (B) must run in time proportional to a polynomial in the number of digits of the input integer N. Secondly, we must also show that (i) A' (B') has a finite failure probability when the input is a composite (prime) number, and (ii) A' (B') certainly decides correctly if N is prime (composite). In 1985, Umesh Vazirani and Vijay Vazirani developed the basic proof technique that helps in establishing the simulation of A (B) by slightly random versions A' (B'). Using this technique, implementation of randomization is finally realized using even slightly random sources.

2.5 Tackle the adversary, expect gains with simple moves

The time required to compute often depends upon the sequence in which the input elements are given. The performance of a sorting algorithm depends on the specific sequence of numbers in the input. The famous *quicksort* algorithm due to Hoare does miserably when you give sorted inputs but runs faster for certain unsorted inputs. It is however reassuring that the average performance of quicksort is quite good, $O(n \log n)$ for inputs of n distinct numbers. This is true when we assume that all possible $n!$ permutations of n numbers are equally likely. In practice, this assumption may not be applicable. In any case, we do not even need to bother about the sequence in which the input is given. All we need to worry about is how to finally churn out the sorted sequence. Randomization can come to our aid. Suppose we randomly permute the input sequence. Then we can make sure that the behaviour of the rest of the steps are not influenced by the adversary, who provides the input numbers in a sequence of his liking. We can therefore vanquish evil designs of the adversary, whose sole aim could have been to provide the unsorted input in certain specifically difficult sequences. Randomly permuting the input has some drawbacks too. Sometimes, even harmless sequences may be transformed into unfavourable sequences. The advantage is that no matter what inputs are given, all possible sequences of the input numbers are equally likely to result from the random permutation step. Therefore, the rest of the steps in the sorting algorithm could have a good expected behaviour. Indeed, sorting can be done in *expected* $O(n \log n)$ time. Certain runs on any input (with the input arbitrarily permuted) could still require a worst case of $O(n^2)$ time. The average running time over all possible random permutations for the same input is $O(n \log n)$, the expected running time of the sorting algorithm.

In addition to sorting the input, it is also possible to build a binary search tree (called a *randomized binary search tree*) with the n input numbers. This tree helps in searching for a number's existance in the tree, a query that is so common in practice (if the numbers were roll numbers of students taking a certain course, we would like to know whether a student with an arbitrary roll number is doing the course). This tree is built with a randomized permutation of the input and is not a balanced search tree like an AVL tree or a B+ tree. We therefore avoid the trouble of balancing the tree with the insertion of every element. At the same time, it can be shown that the expected depth of the tree would be $O(\log n)$, which is also the bound on the depth of a perfectly balanced tree. The expected depth is the average depth over trees that would be built for the same input, by permuting the input in all possible ways.

This paradigm of randomized computing is called *incremental construction*. The same technique of an initial random permutation is used extensively in geometric computation for constructing various structures such as planar maps (with applications in Geographical Information Systems), convex hulls (in two and higher dimensions), Voronoi diagrams and so on. The inputs in these geometric algorithms are essentially sets of points or line segments. The input set is first permuted randomly to vanquish the adversary. The rest of the steps are also simple; they happen to be based on very simple *data structures* [7]. Algorithms that use simple data structures are easier to implement. On the whole, algorithms that do not use randomization may sometimes become very complex and difficult to implement. If such algorithms are required to perform efficiently in the worst case, they may need complex logic and data structures. For instance, the first deterministic (not randomized) algorithm (due to Chazelle and Edelsbrunner in the late eighties) that reported intersections of any set of n line segments in the plane, used complex data structures and highly sophisticated techniques to achieve a running time of $O(n \log n + s)$ in the worst case, where s is the number of intersecting pairs of segments. On the other hand, Mulmuley's randomized algorithm for the same problem belongs to the incremental construction paradigm where the n line segments are randomly permuted and inserted one by one into certain structures, detecting intersections on the fly. Not only are the data structures simple, the whole algorithm is also quite straightforward and easy to implement. A little bit of analysis is used to show that the expected running time of Mulmuley's algorithm is $O(n \log n + s)$, matching the worst case running time bound of Chazelle and Edelsbrunner's algorithm.

2.6 Cheap, simple, smart and sure ways to divide and rule

In many applications, we need to search within data structures to answer queries of membership. Such a problem was mentioned in Section 2.5 where a randomized binary search tree was used to search (for a query number) in a set of n numbers. Searching can also be done using a search structure built using a *randomized divide-and-conquer* approach. We use a constant number r of the n elements to split the set of n numbers into $r + 1$ buckets. The r values are sorted in constant time (yielding a sorted sequence say R) and used to split the n numbers into $r + 1$ buckets; the ith bucket has elements whose values are between the ith and the $(i - 1)$th values in the sequence R, and the last bucket has elements greater than the largest value in R. The

[7] Data structures are *gadgets or components* used by algorithm designers to efficiently and conveniently store and process data, leading to efficient and elegant solutions. Efficient data structures help us design fast algorithms. The *binary heap* data structure yields an optimal $O(n \log n)$ time algorithm for sorting n numbers. The *fibonacci heap* is used to run Dijkstra's single source shortest path algorithm in spectacular $O(n \log n + e)$ time for a graph with n vertices and e edges.

r numbers in R are chosen using *random sampling*, so that all r-subsets of the n numbers are equally likely to be chosen. It is easy to see that one can search and find the bucket (possibly) containing an arbitrary query number within the $r+1$ buckets in constant time. The sampling of the r numbers is crucial; it is done so that the size of the maximum bucket is bounded suitably. Limiting the size of buckets helps in establishing efficient searching operations. Each of the $r+1$ buckets is further split recursively in similar manner as long as its size is above a constant $b > r$. Buckets with less than b elements are made leaf nodes. So, we get a search tree where the size of each bucket is limited by a bound with respect to the size of its parent bucket in the tree. Using such a scheme, an $O(\log n)$ time searching operation can be supported by the tree. The randomized step of splitting the root of the tree (the set of the given n numbers) into the $r+1$ buckets is possible in expected $O(n)$ time. Recursively building the splits at these buckets and their children and so on, requires expected $O(n \log n)$ time. Thus the entire (r+1)-ary search tree is built in expected $O(n \log n)$ time.

Random sampling is used in many geometric algorithms for building efficient search structures for answering queries. One good example is the data structure built using this technique for answering *point location queries* in an arrangement of lines in the plane. Random sampling is also used for triangulating polygons in efficient expected running time; Clarkson, Tarjan and Wyk showed in 1989 that an n-sided simple polygon can be triangulated in expected $O(n \log^* n)$ time. (If we repeatedly take the logarithm of n for k steps and the value falls down to 1, we say that $\log^* n = k$. This is a very slowly growing function and is practically almost a constant for not too large n.)

3 Back to nature

We know that $NC \subseteq P \subseteq NP$. It is not known whether any two of these classes are equal. For each problem in P, there exists an algorithm that runs in time proportional to a polynomial in the size of the input to the algorithm. Such a known algorithm can very well be implemented on a usual computer. Given an arbitrary problem from the class NP, we can certainly solve it on any computer by running a brute-force guess and check method. A typical problem is the *graph isomorphism* problem where two n-vertex graphs are given. The vertices of both graphs are numbered arbitrarily and the vertex pairs making the edges are listed for each of the two graphs. We have to find whether the two graphs are *isomorphic*. This means we have to determine whether it is possible to permute or renumber all vertices of one of the graphs so that the two sets of edges become identical. In other words, we have to find a permutation p such that vertices i and j are connected in the first graph if and only if vertices $p(i)$ and $p(j)$ are connected in the second graph. If the adversary provides a permutation, it is a simple matter to check whether the edge sets coincide after applying the permutation p to the second graph. If the check succeeds, the graphs must be isomorphic. If the adversary is absent, we will have to check all possible permutations of n numbers to determine whether the graphs are isomorphic. This requires time proportional to an exponential function of n. With parallelism provided by an exponential number of processors and suitable synchronization between processors, we can solve the problem in polynomial time. The problem however is the practicability of such a computer that provides such massive parallelism. Whether we resort to electronics, optics or molecules, such exponential scale of parallelism does not seem feasible in the physical world. For graph theoretic, number theoretic or optimization problems of considerable input sizes, the number of processors required would become astronomical.

All hope should not be lost though; people have already studied newer and varied models of parallel computing. One such model is that of *quantum computing*, where the degree of parallelism is apparently not so constrained as in electronic parallel computers. Polynomial time is sufficient in the quantum computing model to solve certain very important problems. Shor proved recently that *factoring*[8] and *discrete logarithms*[9] problems can be solved in polynomial time on quantum computers. (We omit the very technical definitions of these special functions here.) These problems remain unsolved in the usual reasonable computing paradigm;

[8] The RSA public key cryptosystem can be attacked by using algorithms to factor the product of two large primes.

[9] Let $y = g^x \bmod p$. The discrete logarithm problem is that of computing x, given y, g and a prime p.

the hard problems of *factoring* and *discrete logarithms* are not known to be solvable in polynomial time. The possible intractability of these two problems forms the basis for secure methods in modern day cryptography. Actual implementations of quantum computers could therefore render certain current cryptographic solutions insecure.

How far can quantum computing solve our known hard problems? The answer to this question is not very straightforward. Recent research reveals that problems from the class BPP [10], that can be solved using randomized algorithms in polynomial time (with bounded error) can also be solved in polynomial time using quantum computations (with bounded errors). The class of decision problems solvable on quantum computers with finite bounded errors in polynomial time is called BQP. The proper containment of BPP in BQP is however a major open question. A separation here would show the supremacy of quantum computing over randomized computation. Such fundamental separation results are very rarely resolved in computer science. Outstanding examples are similar harder questions like the equality of P and NP (open since the formal inception of NP into computer science literature in the early seventies). It is easy to see that $P \subseteq BPP$. The equality here is not established, neither ruled out. The $P = BPP$ is another outstanding and fundamental open question in computer science. There is a feeling that BPP after all is not very powerful. It does not appear to include many hard problems and very few algorithms, if any, use the full power of BPP. Moreover, BPP can be *derandomized*[11] to be captured into a very low level (second level) in what is known as the *polynomial hierarchy* PH[12]. So, if $NP \subseteq BPP$, then the whole of PH would collapse into BPP, something that would be rather surprising. Irrespective of the power of BPP, we must be concerned about the possibility that $NP \subseteq BQP$. Researchers have established results that demonstrate the difficulty of simulating NP with BQP.

We conclude by mentioning a few more results. It is possible to run some exponential time computations in polynomial space, by reusing space. BQP can be simulated not only in classical exponential time but also in classical polynomial space. A most important result is that BQP is a subset of $P^{\#P}$. This essentially is the class of computations done in polynomial time with cost free access to query questions of the $\#P$ kind. Intuitively and very informally, $\#P$ is certainly a very powerful set of functions where a function $f(x)$ returns (in binary), the number of solutions y that satisfy the input x, where y's length is bounded by a polynomial in the length of x and the verification of y satisfying x is also possible in polynomial time. So, the power of polynomial time quantum computations as in BQP is restricted by a polynomial number of free queries to counting problems of polynomially checkable solutions. This limits the power of quantum computing.

---

[10] RP is a subset of the important and mysterious class of problems called BPP. Informally, BPP is the class of decision problems that are solvable in polynomial time using randomization with error probability 1/3. The error probability is finite for both 'yes' and 'no' answers unlike the case with RP.

[11] Derandomizing is simulating the behaviour of a randomized computation using non-randomized computations.

[12] Very informally, the polynomial hierarchy PH is the infinite union of sets from a countable number of levels, where each level has classes with polynomial computing time aided with free access to the power of the previous level in the hierarchy. We refer the interested reader to texts and omit a precise definition here.