Estimation of the Speedup of Distributed Applications

Włodek M. Zuberek

Abstract— **Speedup is one of the main performance characteristics of distributed applications. It is defined as the ratio of application's execution time on a single processor to the execution time, of the same workload, on a system composed on N processors. This paper analyzes, in very general terms, the speedup that can be achieved in distributed environments and shows why some applications scale very well with the number of processors while others have strict limitations on the speedup that can be achieved in distributed environments. The existence of such limitations simply means that a straightforward distribution of a (sequential) workload is not a satisfactory approach, and new algorithms are needed to use distributed environments in a more satisfactory way.**

Keywords— **Distributed systems, speedup estimation, computation-to-communication ratio, iterative methods, state space generation, SETI@home.**

I. Introduction

The performance of computer systems can be increased either by increasing the performance of uniprocessors, or by increasing the number of processors in a (concurrent) system. The current computer technology favors concurrent systems because they are more economical [7]. Moreover, distributed systems are often considered as a less expensive and more easily available alternative to parallel systems [15]. The Beowulf cluster [16], [18] is probably the most popular example of a system composed of (many) standard (or "off-the-shelf") components, connected by a communication medium that exchanges messages among the components of the system. Distributed systems can be tightly coupled, with a high–performance interconnecting network, or can be loosely connected by a local area network or even Internet [5].

As the CPU performance and communication bandwidth increase, distributed computing is becoming an attractive platform for high–performance computing. Although the number of practical applications of distributed computing is still somewhat limited [6] and the challenges – in particular, the standardization – are still significant, there are some spectacular examples of using thousands and even millions of processors working in a coordinated way on the same problem.

In distributed applications, the total workload is divided among the processors of the system. One of the main performance characteristics of a distributed application is its speedup [17], which is usually defined as the ratio of the execution time on a single processor, $T(1)$, to the execution time of the same workload on a system composed of N processors, $T(N)$:

$$S(N) \;=\; \frac{T(1)}{T(N)}.$$

The speedup depends upon a number of factors which include the number of processors and their performances, the connections between the processors, the algorithm used for the distribution of the workload, etc. Some of these factors may be difficult to take into account when estimating the speedup of a distributed application. Therefore, in many cases, a simplified analysis is used to characterize the steady–state behavior of an application. This simplified analysis is based on a number of assumptions, such as a uniform distribution of workload among the processors, constant communication times, and so on.

This paper estimates the speedup of several distributed applications showing that for some applications the performance increases linearly with the number of available processors, while other applications exhibit strict limitations on the speedup that can be achieved in distributed environments.

II. Large Sparse Linear Systems

The solution of large, sparse systems of linear equations is a common part of many computational techniques. Large sparse systems of linear equations are created by the finite element method, they arise in analysis of electronic circuits (in the frequency domain), in the solution of Markov chains, and many others. For large sparse systems of equations, iterative methods [3], [4], [8] are often preferable to direct methods because they are less demanding with respect to memory requirements and have low computational demands per iteration, their convergence, however, can be a challenge.

For distributed processing, the (numerous) equations are split into approximately equal groups allocated to different processors. The iterative process repeatedly executes the following three consecutive steps:

1. the current approximation to the solution is distributed to all processors,
2. parts of the new approximation to the solution are determined (concurrently) by the processors,
3. the new approximation is collected from all processors and the convergence is checked.

The first step typically uses a broadcast (or multicast) operation, and it can be assumed that its execution time, T_b, does not depend upon the number of processors (in fact,

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through Grant RGPIN-8222.

it usually depends on this number, but this dependence is ignored here as it is rather inessential). The last step requires a transfer from all processors to a single processor which performs the convergence check, so the transfers are performed sequentially. It is assumed that the total execution time of this step is equal to $N * T_r$, where T_r is the communication time of a single processor. Although T_r depends upon N, the dependence is not very strong [14], and is neglected here.

Let T_s denote the total (sequential) computation time of one iteration. The (approximate) time of a distributed execution of a single iteration is then:

$$T(N) \ = \ T_b + T_s/N + N * T_r.$$

For simplicity, it can also be assumed that $T_r = T_b$, which is an oversimplification, but not very significant, as it appears. With this additional assumption, the speedup is:

$$S(N) \ = \ \frac{T_s}{T_s/N + (N+1) * T_r}.$$

Let $r_{comp/comm}$ denote the ratio T_s/T_r, i.e., the ratio of total (sequential) computation time (per iteration) to the communication associated with a single processor (such a ratio makes sense only for programs with cyclic behavior). Then the speedup becomes a function of two variables, N and $r_{comp/comm}$:

$$S(N) \ = \ \frac{r_{comp/comm}}{r_{comp/comm}/N + N + 1}.$$

Fig.1 shows the values of $S(N)$ for $N = 2, ..., 20$ and for $r_{comp/comm} = 10, ..., 100$.

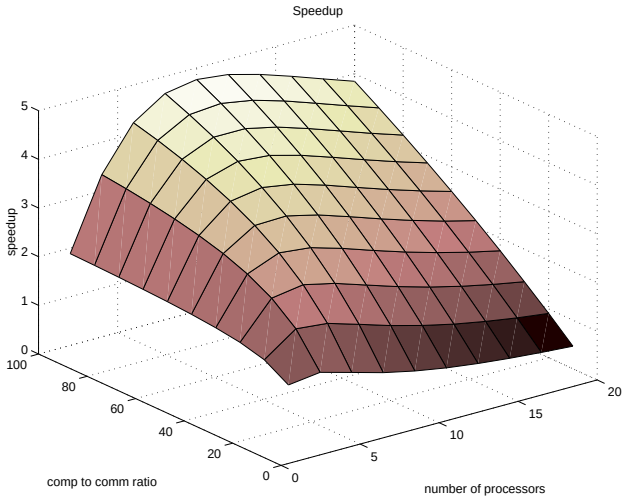

Fig.1. Speedup of a distributed iterative solver.

Reasonable speedups (around 5) can be obtained only when the computation–to–communication ratio is sufficiently high. The best speedup is obtained for a rather small number of processors (5 to 10); for a larger number of processors, the execution time actually increases as it is dominated by the communication time.

It should be observed that the simplifying assumptions are not really important as they affect terms which do not have significant influence on the speedup, especially for larger values of N.

For this applications, when the number of processors is large, the communication time becomes the dominating component of the execution time. In order to reduce the communication time, the results can be collected in a hierarchical way, in which first the results of computations are collected in groups of, say, K processors, and then the results of groups are combined together. It can be shown that the number of groups that minimizes the total communication time is equal to $\sqrt{N}$, and then the speedup becomes:

$$S(N) \ = \ \frac{r_{comp/comm}}{r_{comp/comm}/N + 2 * \sqrt{N} + 1}.$$

Fig.2 shows the values of $S(N)$ for $N = 2, ..., 20$ and for $r_{comp/comm} = 10, ..., 100$ for this modification.

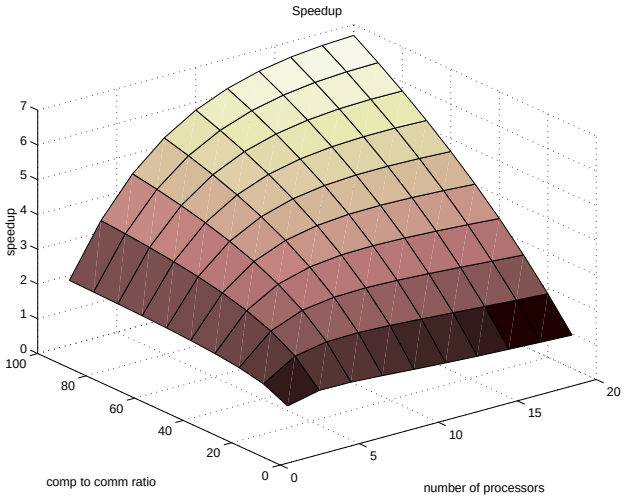

Fig.2. Speedup of a modified iterative solver.

The relation between the speedup and the number of processors is quite different in this case, better speedup values can be obtained then previously, and the speedup is much less sensitive to the number of processors. For very large values of N, further improvement can be obtained by additional levels of the hierarchical collection of results.

III. State Space Exploration

A popular approach to the verification of discrete–state systems is based on the exhaustive exploration of their state spaces to check if all states which are reachable from the initial state(s) satisfy the required conditions (e.g., mutual exclusion, boundedness, and so on). For large net models, such exploration of the state space can be quite time-consuming, and the state space can easily exceed the resources of a single computer system. The availability of distributed systems offers an attractive alternative to the traditional sequential reachability analysis.

In distributed reachability analysis, the (yet unknown) state space is partitioned into n disjoint regions, $R_1, R_2, ..., R_n$, and these regions are constructed independently by n identical processes running concurrently on N

different machines (most often $N = n$). At the end, the regions can be integrated in one state space if needed.

There are several approaches to the distributed reachability analysis. Many results and implementation details for parallel reachability analysis on shared–memory multiprocessors are given in [1], [2]. In shared–memory systems, however, the processors are tightly connected, so inter–processor communication is quite efficient, which makes such systems significantly different from loosely connected collections of workstations or PCs. Distributed state space generation described in [12] is organized into a sequence of phases, and each phase contains processing of currently available states followed by communication in which non-local states are sent to their regions (i.e., processors). The next phase begins only after completion of all operations of the previous phase. The generation of the state space is thus similar to the solution of linear systems discussed in the previous section. Reasonable values of speedup were reported for small numbers of processors (2 to 4) and for large state spaces.

An approach described in [11], [13] is a straightforward modification of the sequential algorithm for state space generation:

```
1. Sequential state space generation:
2. var States := ∅;                          (* set of states *)
3.     unexplored := ∅;                      (* queue of states *)
4. begin
5.     for each s in IntitalStates do
6.         States := States ∪ {s};
7.         insert(unexplored, s)
8.     endfor;
9.     while nonempty(unexplored) do
10.        state := remove(unexplored);
11.        for each s in NextStates(state) do
12.            if s ∉ States then
13.                States := States ∪ {s};
14.                insert(unexplored, s)
15.            endif
16.        endfor
17.    endwhile
18. end.
```

where *IntialStates* denotes the set of initial states, and function *NextStates(s)* determines the set of successor states of state s.

Distributed generation of the state space uses a partitioning function, *region(s)*, which assigns each state s to the region to which it belongs. The detailed definition of this function depends upon the representation of the state [12], [11], but is assumed that this function uniformly distributes the states over the regions (which may not be quite simple to accomplish because the state space is not usually known in advance).

Each of the processors constituting the distributed system analyzes states belonging to one of the regions. For each analyzed state, the set of successor states is determined. A successor state can be in the same region (in which case it is called a *local* state) or in a different region (in which case it is called an *external* state). All external states are sent to regions (i.e., processors) determined by the partitioning function.

In order to perform state analysis concurrently with communication, each processor runs three processes: the *Analyzer*, responsible for processing the states, the *Sender*, responsible for sending messages to other processes, and the *Receiver*, responsible for receiving messages from other processes and for the termination detection [11], [13]. Since the *Analyzer*, *Receiver*, and *Sender* processes for each region reside on the same processor, they communicate by shared variables. The *Receiver* process passes the data (i.e., states) received from other processors to the *Analyzer* process using a shared buffer *recv_buff*, while the *Analyzer* process passes all external states to the *Sender* process using another shared buffer *send_buff*.

Processes residing on different processors constitute a "virtual machine"; they communicate by exchanging messages using a popular message passing library [9].

The distributed reachability analysis is started by creating the *Analyzer* processes and sending the initial states to the corresponding processors:

```
1. Initialization of distributed state space generation:
2. begin
3.     get N and the description of all processors;
4.     for i := 1 to N do
5.         create Analyzer on processor[i]
6.     endfor;
7.     for each s in InitialStates do
8.         send(s, processor[region(s)])
9.     endfor
10. end.
```

Each *Analyzer* processes the states from the internal queue *unexplored* (local states) and from *recv_buffer* (non–local states), with non–local states taking priority over the local ones:

```
1. Analyzer_i:
2. var States_i := ∅;                        (* set of states *)
3.     unexplored := ∅;                      (* queue of states *)
4.     cont := true;                         (* continuation flag *)
5. begin
6.     create Receiver, Sender on this_host;
7.     while cont do
8.         if empty(recv_buffer) ∧ nonempty(unexplored) then
9.             state := remove(unexplored);
10.            new := true
11.        else
12.            state :=get(recv_buffer);
13.            if state = null then
14.                cont :=false
15.            else
16.                new := state ∉ States_i;
17.                if new then
18.                    States_i := States_i ∪ {state}
19.                endif
20.            endif
21.        endif;
22.        if cont ∧ new then
23.            for each s in NextStates(state) do
24.                if region(s) = i then
25.                    if s ∉ States_i then
26.                        States_i := States_i ∪ {s};
27.                        insert(unexplored, s)
28.                    endif
29.                else
30.                    put(send_buffer, s)
31.                endif
32.            endfor
33.        endif
34.    endwhile
```

35. **end**.

A global termination detection algorithm is interleaved with the computations, repeatedly checking if all processors have finished their tasks [11]. When global termination is detected (i.e., when all *Analyzer* processes are waiting on *get* operation – line 12), a special *null* state is sent to all *Analyzer* processes to terminate their operation (lines 13, 14).

Let the (average) time needed for analysis of a single state be denoted by T_a, and the (average) time of sending one state from one processor to another by T_c. For the ideal, uniform distribution of workload among the processors, and uniform distribution of states among the regions, assuming the steady–state conditions in which for each analyzed state there is one new generated state which is local with probability $1/N$ and external with probability $(N-1)/N$, and sequential transmission of messages in the interconnecting network, the speedup can be expressed as:

$$S(N) = \frac{N * T_a}{\max(T_a, (N-1) * T_c)}$$

Let $r_{comp/comm}$ be the ratio of T_a/T_c. Then the speedup becomes a function of two parameters, N and $r_{comp/comm}$, as before:

$$S(N) = \frac{N * r_{comp/comm}}{\max(r_{comp/comm}, N-1)}$$

Fig.3 shows the values of $S(N)$ for $N = 2, ..., 20$ and for $r_{comp/comm} = 1, ..., 25$.

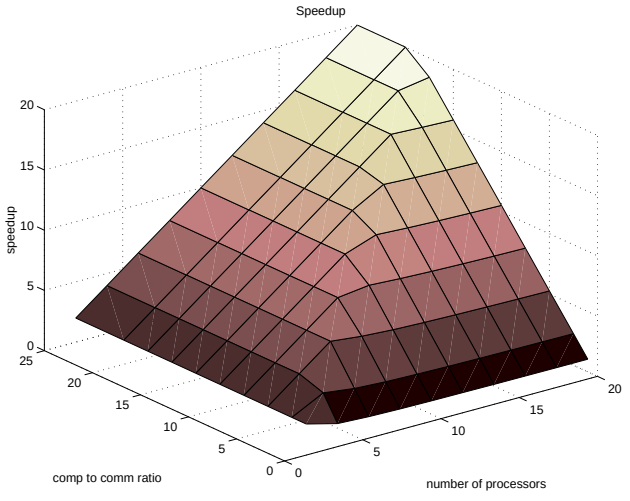

Fig.3. Speedup of a distributed state space generation.

For $r_{comp/comm} > N-1$, the speedup is equal to N, its ideal value, because the communication is being done "in the background" of computing, and it does not affect the execution time. However, when $r_{comp/comm} < N-1$, the speedup degrades linearly with the value of $r_{comp/comm}$; in this case, the performance of the distributed application is determined by the performance of the interconnecting network.

IV. Massively Distributed Computing

The SETI@home project, managed by a group of researchers at the Space Science Laboratory of the University of California at Berkeley, is the first attempt to use large-scale distributed computing to perform a sensitive search for radio signals from extraterrestrial civilizations [19].

SETI@home uses a dedicated L band receiver at the National Astronomy and Ionospheric Center's 305-meter radiotelescope in Arecibo, Puerto Rico. The receiver provides a beam of 0.1 degree width which is digitized and converted to a 2.5 MHz-wide band centered at the 1,420 Mhz hydrogen line. It is believed that this frequency is one of the most likely locations for deliberate extraterrestrial transmissions. Digitized data are recorded continuously on magnetic tapes, along with data on telescope coordinates, time and some additional information. A single tape (35 gigabytes) records 17 hours of observations. Tapes are mailed to Berkeley for analysis; the complete survey requires 1,100 tapes to record the total of 39 terabytes of data. Observations began in October 1998.

At Berkeley, data are divided into small "work units" by first splitting the 2.5 MHz bandwidth into 256 sub-bands (by means of a 2048 point FFT and 256 eight point inverse transforms), an then dividing each 9,765 Hz sub-band into units of 2^{20} samples (which corresponds to 107 seconds of recorded data). Subsequent work units overlap by 20 to 30 seconds to allow full analysis of signals that might be on the boundary of units. Work units are sent over the Internet to the client programs around the world for the bulk of the data analysis. A database of work units, their processing status, and returned results is maintained in Berkeley.

SETI@home distributes client software for about 50 different combinations of processor and its operating system [19]. For Microsoft Windows and Apple Macintosh computers, the software installs itself as a screen saver, processing data only when the computer is not used. For other platforms, it runs in the background and becomes inactive whenever the user executes his jobs.

The client software, after receiving a work unit, performs a baseline smoothing to remove any wideband features (this prevents the client from confusing fluctuations in broadband noise with intelligent signals), and then searches the data for signals with drift rates between -10 Hz/sec to +10 Hz/sec in steps of 0.0018 Hz/sec (to take into account the unknown accelaration of a rotating planet sending the signals). At each drift rate, the client searches for signals at one or more bandwidths between 0.075 and 1,221 Hz. The data are examined for signals that exceed 22 times the mean noise power. All potential signals are sent back to the central server for further processing.

Analysis of a single work unit requires 2.4 to 3.8 trillion floating-point operations (Tflops), and takes 5 to 6 hours of a 1 GHz processor. On average, a client reports 8 detected signals for each work unit.

The vast majority of detected signals are created by sources of narrow-band emissions at or near the observatory, such as equipment, aircraft, satellites, and other transmitters. As these signals are long-duration ones, they

are rejected in post-processing phase [19].

Some of detected signals are caused by errors in computers or communication networks. Although the probability of such errors is very low, the amount of computations required by the SETI@home project (in the order of millions of computation-years) must take the existence of such errors into consideration. Therefore each work unit is processed 3 times, on different processors, and the obtained results are compared for consistency.

The number of "standard", 1 GHz processors, needed for on-line processing of the recorded data can be estimated as follows. Analysis of 80 seconds of data (i.e., 107 seconds with 20 to 30 second overlap) in a single sub-band requires 5 to 6 hours of processing, so on-line processing of a single sub-band requires:

$$\frac{3,600 * 5.5}{80} \;=\; 247.5 \text{ processors.}$$

Multiplying this number by 256 sub-bands, and by the replication factor 3, results in almost 200,000 "standard" processors working "full time" on the analysis of the recorded data. The actual number of computers registered for the SETI@home project is greater at least an order of magnitude ([10] reports 2.5 millions of registered computers), but these computers are usually not available "full time" ([10] reports that only about 20 % of registered computers are active), and not all have 1 GHz processors.

Let the execution time of a single work unit on a "standard" processor be denoted by T_u, and let T_d denotes the time of sending a work unit to a client, while T_r – the time needed to return the results of data analysis. The speedup, $S(N)$, of distributed processing on a system composed of N processors, is:

$$S(N) \;=\; \frac{N * T_u}{T_d + T_u + T_r}$$

and, since T_u is much greater than T_d and T_r:

$$S(N) \;\approx N$$

so, for this particular application, the practical speedup approximates the ideal speedup. This is why the SETI@home project is so spectacularly successful.

V. Concluding Remarks

This paper analyzes, in very general terms, the speedup which can be obtained in distributed environments. Using a number of simplifying assumptions, it introduces several classes of applications and estimates the speedup that can be obtained by distributed execution of an (idealized) application in each class. The paper shows that some classes scale quite well with the number of processors while others impose very strict limitations on the speedup, which simply means that a straightforward distribution of a sequential workload is not a satisfactory approach in such cases, and new algorithms are needed to use distributed environments in a more satisfactory way.

SETI@home may be the most imaginative large-scale distributed application, but there are other applications which could readily benefit from execution on a distributed environment:

- Complex modeling and simulation techniques that increase the accuracy of results by increasing the number of random trials; the trials can be run concurrently on many processors, and the results combined to achieve greater statistical significance.
- Applications that require exhaustive search through a huge number of results that can be distributed over the many processors, such as drug screening [6].
- Simulations of complex systems (such as VLSI designs) in which the system is partitioned into a number of subsystems, and the subsystems are simulated concurrently on different processors.

In the future, traditional distributed systems are expected to be used in specialized domains, such as transaction processing for banking applications, in mainstream applications, however, grid computing is emerging as the next evolutionary platform for large–scale computing.

References

[1] S.C. Allmaier and G. Horton, "Parallel shared-memory state-space exploration in stochastic modeling"; in *Solving Irregularly Structured Problems in Parallel* (Lecture Notes in Computer Science 1253), pp.207-218, Springer-Verlag 1997.

[2] S.C. Allmaier, S. Dalibor, and D. Kreische, "Parallel graph generation algorithms for shared and distributed memory machines"; in *Parallel Computing: Fundamentals, Applications and New Directions* (Advances in Parallel Computing 12), pp.581-588, Elsevier, North-Holland 1997.

[3] O. Axelsson, *Iterative solution methods*; Cambridge University Press 1994.

[4] R.H. Chan, T.F. Chan, and G.H. Golub (eds.), *Iterative methods in scientific computing*; Springer-Verlag 1997.

[5] G. Couloris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design* (2-nd ed.); Addison-Wesley 1994.

[6] L. Erlander, "Distributed computing: an introduction"; Extreme Tech, April 4, 2002.

[7] V.K. Garg, *Principles of distributed systems*; Kluwer Academic Publ. 1998.

[8] A. Greenbaum, *Iterative methods for solving linear systems* (Frontiers in Applied Mathematics 17); SIAM 1997.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. a users' guide and tutorial*; MIT Press 1994.

[10] E. Korpela, D. Werthimer, D. Ansrson, J. Cobb, and M. Lebofsky, "SETI@home: massively distributed computing for SETI"; Computing in Science and Engineering, vol.3, no.1, pp.78-83, 2001.

[11] I. Rada, "Distributed generation of state space for timed Petri nets"; M.Sc. Thesis, Department of Computer Science, Memorial University of Newfoundland, St.John's, Canada 2000.

[12] P. Marenzoni, S. Caselli, and G. Conte, "Analysis of large GSPN models: a distributed solution tool"; Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM'97), pp.122-131, 1997.

[13] I. Rada and W.M. Zuberek, "Distributed generation of state space for timed Petri nets"; Proc. High Performance Computing Symposium 2001, Seattle, WA, pp.219-227, 2001.

[14] M.R. Steed and M.J. Clement, "Performance prediction of PVM programs"; Proc. 10-th Int. Parallel Processing Symposium (IPPS-96), pp.803-807, 1996.

[15] J.A. Stankovic, "Distributed computing"; in *Distributed Computing Systems*, IEEE CS Press 1994.

[16] T.L. Sterling, *How to build a Beowulf: a guide to the implementation and application of a PC clusters*; MIT Press 1999.

[17] B. Wilkinson, *Computer architecture – design and performance* (2-nd ed.); Prentice Hall 1996.

[18] Beowulf home page is "`www.beowulf.org`".

[19] SETI@home home page is "`setiathome.ssl.berkeley.edu`".