ELEN 162
A. I. Zecevic

Lecture Notes for Week 8

Fundamentals of Parallel Computing

In this lecture we will describe some basic characteristics of parallel computers, such as memory architectures, performance criteria and interprocessor communications. We will also consider certain important computational problems whose solution can be significantly accelerated if the process is parallelized. In this context, we will devote special attention to systems of linear equations, since they play a central role in solving a wide range of problems (including nonlinear algebraic and differential equations).

Parallel Architectures

High-performance computers that consist of multiple processing units can be configured in a number of different ways. With that in mind, it is helpful to begin by drawing a distinction between *multicore* and *multiprocessor* architectures. In the multicore case, the different processing units (known as *cores*) are located on a single chip, and are designed to simultaneously execute different instructions of the same program.

A typical configuration of this sort with two levels of cache is shown in Fig. 1. Note that each core has its own L1 cache, which consists of two parts - one that stores the data, and another that contains instructions. The L2 cache has a larger capacity, and is shared by all units.

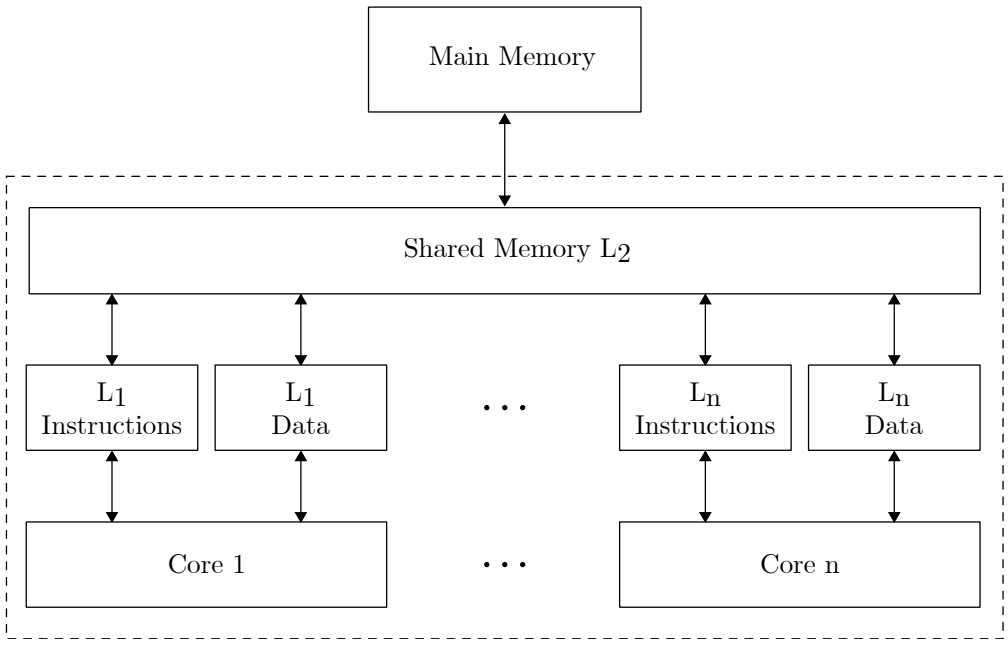

Figure 1: A multicore architecture with two levels of cache.

One of the advantages of such a design is that the cores (and the associated memory units) are physically close to each other, and can therefore be easily connected. This simplifies data retrieval as well, since each core can access its local cache (L1) very quickly, and will send a request to L2 cache only if it cannot find the information that it needs. Such requests can be handled in several different ways, depending on the total number of cores.

Although multicore chips are used in virtually all newer computers (and the number of cores that can be placed on a chip is steadily increasing), in the following we will be more interested in *multiprocessor systems*, since they are better suited for high-performance scientific computing. Such systems consist of multiple CPUs that are part of the same computer, but are not on the same chip. These units are generally more powerful and versatile, and can simultaneously execute *different* programs.

The memory organization in such systems can take one of two basic forms, which differ in the amount of storage space that is directly accessible to each processor.

Shared Memory Architecture

Figure 2 shows an example of a *shared memory* architecture where each processor has its own L1 cache, but the main memory is common to all of them.

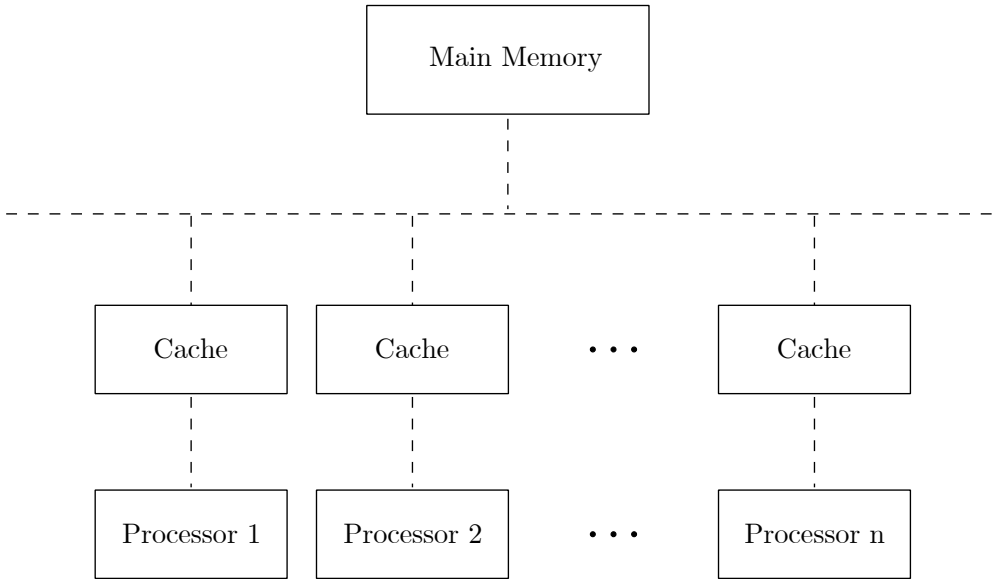

Figure 2: Example of a shared memory architecture.

This type of configuration eliminates the need for direct communication links between processors, since they can easily exchange information by writing to (and reading from) the main memory. The difficulty, however, is that such exchanges are susceptible to delays, since access times to the main memory can vary significantly. To see why this is so, suppose that one of the processors needs a block of data in order to execute the next step of its computation. If the data is already in the local L1 cache, it can be retreived easily. If this is not the case, however, the processor must request it from the main memory, at which point two possible scenarios can arise.

Scenario 1. The latest version of the data could be available in the main memory. Before it can be fetched, however, the processor must free up the necessary space in its L1 cache. This means that some data (preferably a block that has not been used for a while) must be sent back to the main memory.

Scenario 2. The latest version of the data could reside in the cache of another processor. In such cases, the data would first have to be written into the main memory, and erased from the cache of the processor that modified it. Only then can the steps described in Scenario 1 be executed.

What makes this process potentially unpredictable is the possibility that multiple processors could simultaneously try to update the same block of data. In order to avoid conflicts, this must be done sequentially, so queuing delays are inevitable. In such cases, we must also ensure that successive modifications of the data do not result in any inconsistencies, which further prolongs the time during which the system remains idle. Bottlenecks of this sort can slow down the computation significantly, particularly when the number of processors is large.

Distributed Memory Architecture

A schematic description of a distributed memory architecture is shown in Fig. 3. In this configuration, each processor has its own local memory, and exchanges information with other units through a communication network.

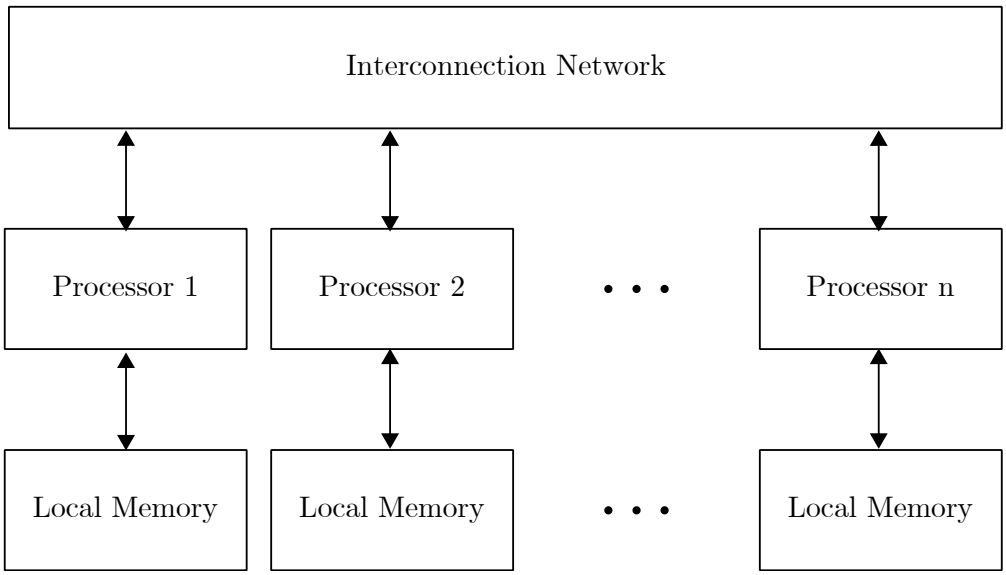

Figure 3: Example of a distributed memory architecture.

It is not difficult to see that this form of organization is considerably more flexible than the one shown in Fig. 2, since the processors do not compete for memory access. Such an arrangement is more costly to implement, however, and requires specialized protocols for interprocessor communication (which we will discuss shortly).

Performance Criteria

In order to evaluate the efficiency of a parallel algorithm, we need to determine the number of tasks that it can perform simultaneously, and account for possible delays that can arise in this process. Two performance measures that are particularly useful for this purpose are *scalability* and *speedup*.

Scalability

Parallel computers that consist of a large number of processors are said to be *massively parallel*. Whether or not such a configuration will be effective depends to a large extent on the nature of the problem, and how well a particular algorithm can exploit its structural features. Given a problem whose size is fixed, we will say that the parallel algorithm designed to solve it is *scalable* if its efficiency increases with the number of processors. Such algorithms are usually well suited for massively parallel computers, since they can utilize a large number of processors (ideally, they could perform the required computations p times faster using p processors).

Speedup

Another measure for the effectiveness of a parallel algorithm is the speedup that it achieves. This quantity can be defined as

$$S_p = \frac{T_s}{T_p} \tag{1}$$

where T_s denotes the time needed to execute the algorithm using a *single* processor, and T_p represents the time needed to do this in parallel using p processors. In practice, S_p is always smaller than p, because parallel computation necessarily entails a certain amount of overhead. As a result, only a portion of T_p is actually devoted to "mathematically useful" activities.

To see how the overhead can be reduced, we will need to briefly consider some of its principal sources. One of them has to do with *load balancing*, which measures how evenly the computational effort is distributed across the processors. When load balancing is inadequate, some processors will inevitably take longer to complete their tasks, and delays are likely to occur. This effect is particularly pronounced if the tasks are *synchronized*, in which case a number of processors could remain idle for prolonged periods of time.

A typical example of a this sort are iterative algorithms in which the next iteration cannot begin before all processors have completed the previous one (and have exchanged the relevant data). We will examine a number of such methods later, but for now it suffices to observe that sychronization adds to the overhead, since it requires special routines that coordinate the activities of different processors. The time needed to execute them can be treated as a form of delay, since no new computations are performed until all the necessary steps have been completed.

Interprocessor Communication

In distributed memory architectures, the speedup is also affected by the time needed for interprocessor communications. It is obviously desirable to minimize this type of overhead

as much as possible, since that can significantly reduce the overall execution time. To explain how this can be done, we must first say a few words about the structure of the messages that are being exchanged, and the way that different processor configurations handle certain basic communication tasks.

How Messages are Structured

We begin by observing that each message consists of one or more *packets*, which can be viewed as groups of bits. Most of these bits represent data that is part of the computation, but a fraction of them is always dedicated to other tasks (such as control, addressing, error checking, etc.). Throughout this discussion we will treat packets as the basic "units" of communication, and will assume that each one of them requires time T_0 to be transmitted across a single communication link. The value of T_0 includes the time needed to assemble a packet and add the necessary control and address bits, possible queing delays, as well as propagation time.

It is important to keep in mind in this context that individual messages are often partitioned into *multiple* packets. As a result, it will generally take longer than T_0 for a processor to receive the entire message from one of its "neighbors". To get a sense for why partitioning is potentially advantageous, consider a scenario where processor i needs to send a message which consists of n different packets to processor j. If we assume that the shortest path between these two processors involves k links, the message would reach its destination at time $t_S = knT_0$ if it were transmitted *as a whole* (since the transmission time is proportional to the number of packets that it contains).

The situation is quite different, however, if we choose to transmit *one packet at a time*. If the first packet is sent at time $t_1 = 0$, the second at time $t_2 = T_0$ and so on, the last packet will obviously leave processor i at time $t_n = (n-1)T_0$. This packet will reach processor j at time

$$t_M = t_n + kT_0 = (n + k - 1)T_0 \tag{2}$$

which implies that $t_M < t_S$. The difference between t_M and t_S can be very significant if n and/or k happen to be large, so breaking up messages into their constituent parts (and sending them in stages) is clearly a useful strategy.

Types of Message-Passing Architectures

When dealing with message-passing architectures, it is important to estimate how long it takes to execute certain basic communication tasks, and how one can optimize this process. It is also important to determine how this delay depends on the number of processors (which we will denote by p).

Processors can be connected in a number of different ways, some of which are illustrated in Figs. 4-8. For the sake of expediency, in the following we will focus exclusively on the so-called *hypercube topology*, since it provides the most flexibile framework for information exchange. In a hypercube, each processor is assumed to have d "neighbors" which are connected to it directly. Figure 7 corresponds to the case when $d = 3$, and Fig. 8 shows what such an arrangement looks like for $d = 4$. It is not difficult to recognize from these two diagrams that a "d-cube" consists of exactly 2^d processors.

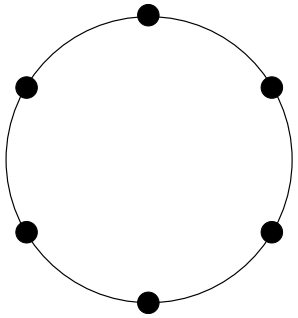

Figure 4: Example of a ring topology.

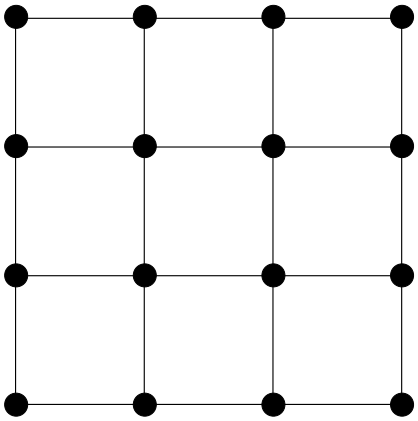

Figure 5: Example of a mesh topology.

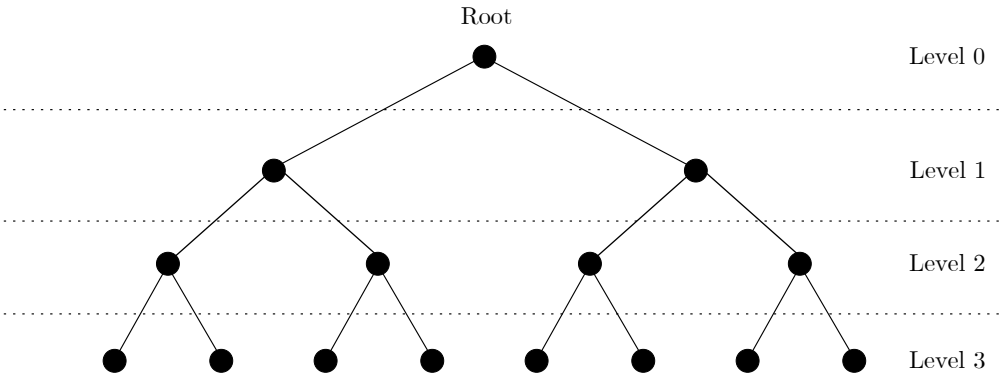

Figure 6: Example of a tree topology.

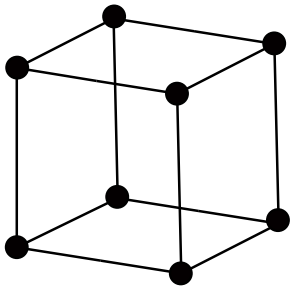

Figure 7: A hypercube with $d = 3$.

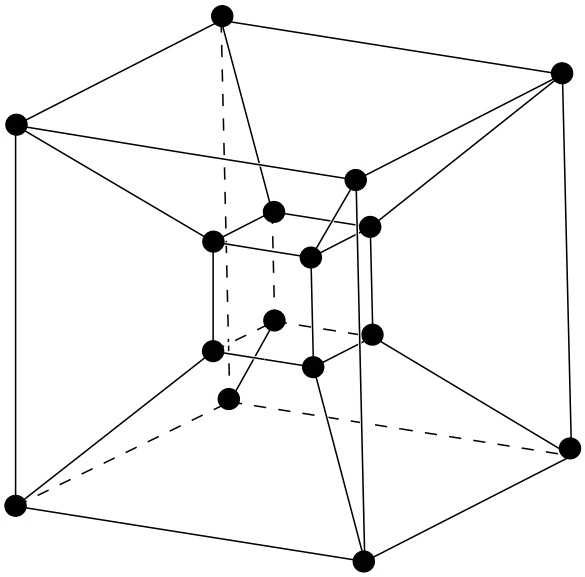

Figure 8: A hypercube with $d = 4$.

Basic Communication Tasks

The simplest communication task in parallel computing is a *single node broadcast*, where an individual processor sends the *same* packet to all others. Sending a *different* packet to each processor gives rise to a more general problem, which is commonly referred to as a *single node scatter*.

When all processors simultaneously perform a single node broadcast, we have what is known as *multinode broadcast*. This task obviously poses some additional challenges, since there is a distinct possibility that different packets will compete for the same communication links. In order to resolve this problem, it is necessary to develop scheduling protocols that can minimize queuing delays.

An even more complicated scenario corresponds to the case when each processor simultaneously performs a single node scatter. This task (which is referred to as a *total exchange*) can give rise to significant communication delays if it is not executed carefully.

Remark 1. The first three types of exchanges that we described can be performed "in reverse", if we assume that processors *receive* data instead of sending it. These tasks (which are known as *single node accumulation*, *single node gather* and *multinode accumulation*, respectively) will not be considered in the following, since they are dual to the ones that we will analyze.

When examining how communication tasks can be optimized, it is helpful to represent processors as nodes in a graph. If we have 2^d processors at our disposal, it will be convenient to number the nodes in binary form and link them in such a way that neighboring nodes differ from each other by a *single bit*. By doing so, we can ensure that every node has exactly d neighbors (which is the defining characteristic of a hypercube topology). Fig. 9 shows how this works for $d = 3$.

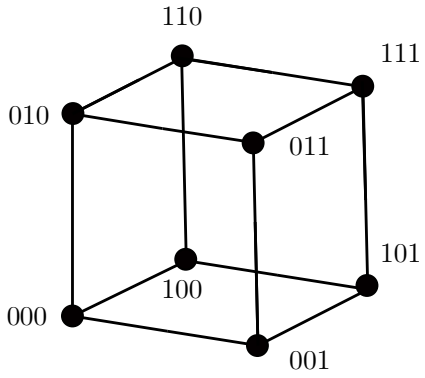

Figure 9: Node numbering for a hypercube with $d = 3$.

We now proceed to describe how a single node broadcast can be implemented on a hypercube. To begin with, we should observe that any node in the hypercube can serve as the *root of* a *spanning tree*. A spanning tree is defined as a subgraph which connects all the nodes using the smallest possible number of edges. It can be shown that this number will always equal $n - 1$ in a connected graph with n nodes.

To demonstrate how spanning trees can be formed, let us assume that the binary index of the root node is $(0, 0, \ldots, 0)$. Our first step will be to define the nodes shown in Fig. 10 as its "children" since their indices differ from $(0, 0, \ldots, 0)$ by a single bit.

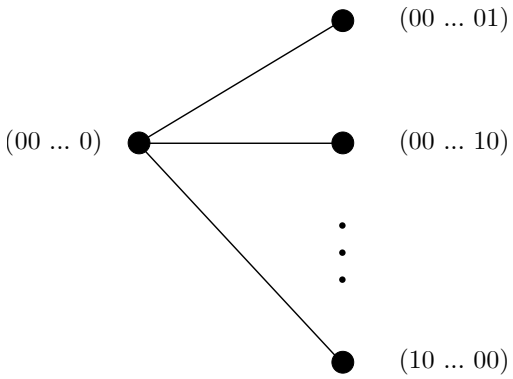

Figure 10: Node $(0, 0, \ldots, 0)$ and its "children".

Each node in Fig. 10 will have "children" of its own, which are obtained by flipping a single 0 into a 1 *following the last* 1 *in its binary index*. This process is illustrated in Fig. 11, which shows the "children" of nodes $(0, 0, \ldots, 0, 1, 0)$, $(0, 0, \ldots, 0, 1, 0, 0)$ and $(1, 0, \ldots, 0, 0)$. It should be noted that node $(0, 0, \ldots, 0, 1, 0)$ has only one of them, $(0, 0, \ldots 0, 1, 0, 0)$ has two and $(1, 0, \ldots, 0)$ has as many as $d - 1$.

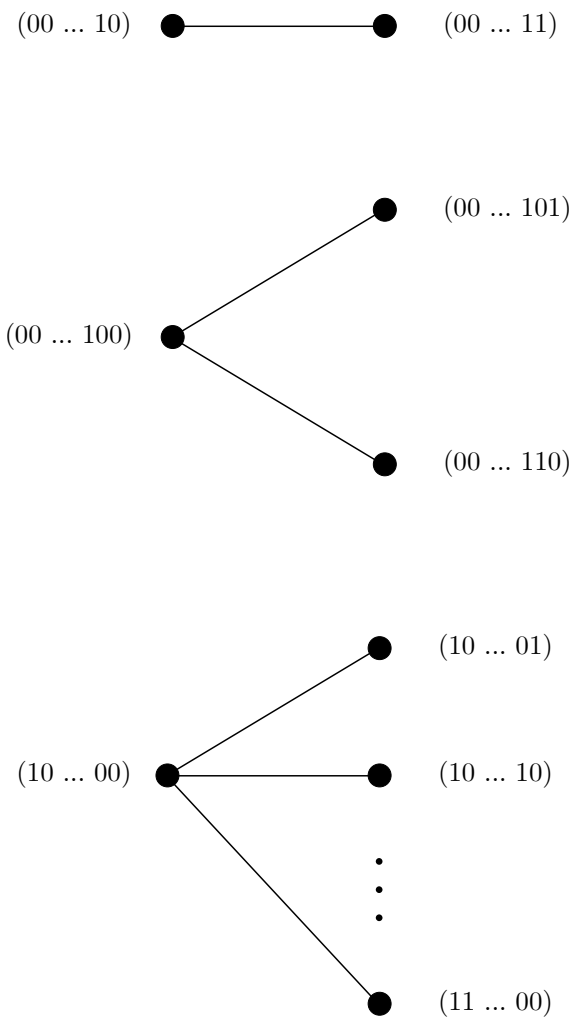

Figure 11: The next step in forming a spanning tree.

By proceeding in this manner, we can systematically form a spanning tree that is rooted in node $(0, 0, \ldots, 0)$. The diagram in Fig. 12 illustrates what such a tree looks like for $d = 3$. The advantage of numbering the nodes in this manner is that it precisely specifies where a packet should be sent from any given node. If it happens to be at node 100, for example, we know that it should be forwarded to nodes 101 and 110 (which are its "children") in the next step. This means that we can formulate an appropriate scheduling algorithm by simply following the graph shown in Fig. 12.

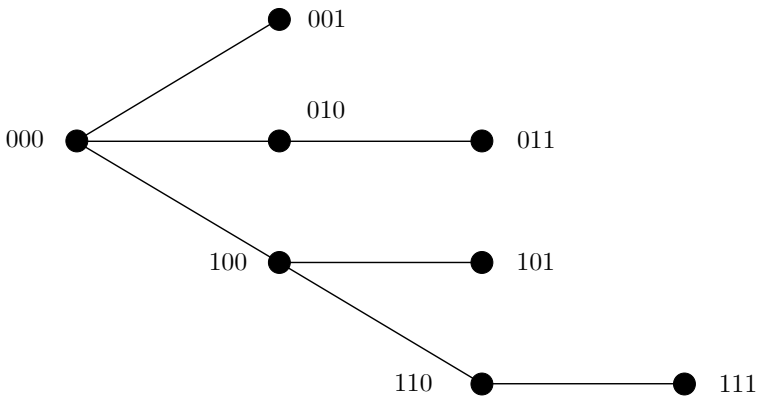

Figure 12: The spanning tree for $d = 3$.

Given that the indices of any two nodes in a "d-cube" differ by no more than d bits, we know that a packet sent by the root node can get to any other node in at most d steps. Recalling that $d = \log_2 p$ in a hypercube, we can conclude that the maximal time for executing a single node broadcast on this type of message passing architecture is

$$T(p) = T_0 \log_2 p \tag{3}$$

In this expression, T_0 once again denotes the time needed to transmit a packet across a single communication link.

We will not analyze the other three communication tasks in detail, but it is helpful to know how their execution time depends on the number of processors. The following table provides this information for a hypercube configuration.

Communication Task	**Execution Time**
Single node broacast	$T(p) \sim \log_2 p$
Single node scatter	$T(p) \sim p/\log_2 p$
Multinode broadcast	$T(p) \sim p/\log_2 p$
Total exchange	$T(p) \sim p$

Table 1. Execution time for different communication tasks.

Examples of Parallelizable Computations

Vector and Matrix Operations

To get a sense for the potential benefits of parallel computing, it is helpful to consider how it can accelerate some basic operations involving vectors and matrices. The simplest example of this sort is the scalar product of vectors x and y

$$x^T y = \sum_{i=1}^{n} x_i y_i \tag{4}$$

which requires n multiplications and n additions. If we were to perform this task serially for two $n \times 1$ vectors, the execution time would be

$$T_S = 2n\Delta t \tag{5}$$

(where Δt denotes the time needed for a single addition or multiplication).

The diagram in Fig. 13 illustrates how this operation can be parallelized for two 4×1 vectors using 4 processors. It is readily observed that this requires an initial step where pairs (x_i, y_i) are multiplied, and 2 subsequent steps to form the overall sum.

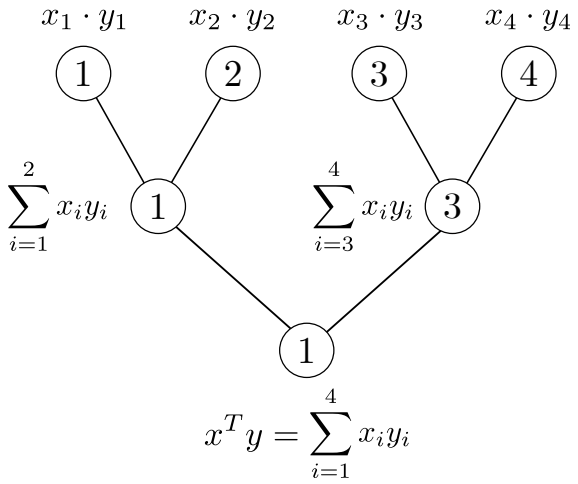

Figure 13: Computing the scalar product using 4 processors.

Given that these operations are performed simultaneously in each step, the total computation time is

$$T_{comp} = 3\Delta t \tag{6}$$

If we assume that the time needed to transmit a number from one processor to another is T_0, the total communication overhead will be $T_{comm} = 2T_0$, and the time required for computing $x^T y$ in parallel becomes

$$T_P = 3\Delta t + 2T_0 \tag{7}$$

This is obviously significantly shorter than the time needed to perform such a computation in series (which is $T_S = 8\Delta t$ in this case).

Figure 14 extends this approach to the case when x and y have dimension 8×1, and 8 processors are available. From this diagram, it is obvious that we now need 3 different steps beyond the initial multiplication, which means that

$$T_P = 4\Delta t + 3T_0 \tag{8}$$

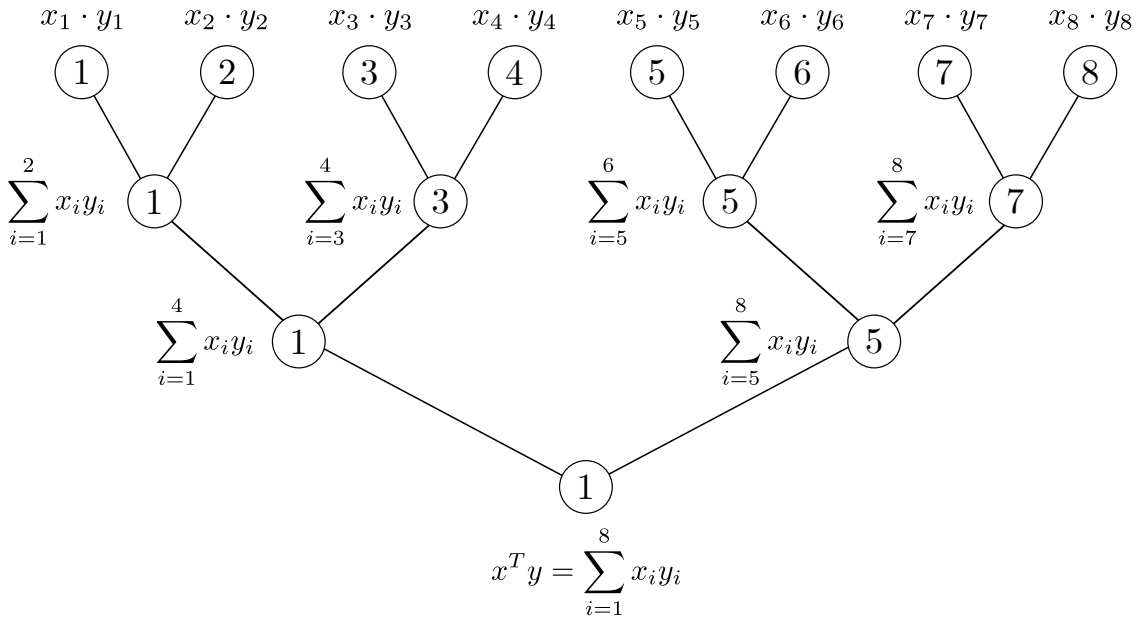

Figure 14: Computing the scalar product using 8 processors.

Figures 13 and 14 suggest that the number of steps needed to compute the scalar product of two $n \times 1$ vectors is $\log_2 n + 1$ if n processors are available. If we include communication overhead as well, the total execution time becomes

$$T_P = [\log_2 n + 1]\Delta t + T_0 \log_2 n \tag{9}$$

This expression allows us to estimate the speedup as

$$\frac{T_S}{T_P} = \frac{2n\Delta t}{[\log_2 n + 1]\Delta t + T_0 \log_2 n} \tag{10}$$

when n processors are used. If we additionally assume that $T_0 = \alpha \Delta t$ (where $\alpha < 1$), we obtain

$$\frac{T_S}{T_P} = \frac{2n}{[(1+\alpha)\log_2 n + 1]} \tag{11}$$

which indicates that the speedup is roughly proportional to $n/\log_2 n$.

The ideas outlined above can be easily extended to matrix multiplication. Given two matrices A and B of dimension $n \times n$, we know that term (i, j) in matrix $C = AB$ can be computed as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{12}$$

Since calculating c_{ij} involves the scalar product of two $n \times 1$ vectors, we know that we can perform this operation using n processors in time

$$T_P = [(1+\alpha)\log_2 n + 1]\Delta t \tag{13}$$

Observing that there are n^2 such elements, it follows that we would need n^3 processors to compute them all simultaneously. A serial algorithm, on the other hand, would require time

$$T_S = 2n^3 \Delta t \tag{14}$$

so the speedup in this case could be as high as

$$\frac{T_S}{T_P} = \frac{2n^3}{[(1+\alpha)\log_2 n + 1]} \tag{15}$$

Remark 2. It is not difficult to see that parallelization algorithms of this sort are highly scalable. In the case of matrix multiplication, for example, they allow us to use as many n^3 processors, which can be a very large number. We should also note that such algorithms can be easily adapted if the number of available processors happens to be smaller.

Linear Algebraic Equations

When discussing the potential advantages and disadvantages of parallel computing, it is important to keep in mind that many important problems *do not* lend themselves to simple parallelization schemes. We will examine a number of such problems in subsequent lectures, but before we do that, we should point out that most of them ultimately reduce to solving systems of the form

$$Ax = b \tag{16}$$

where A is an $n \times n$ matrix, and b is an $n \times 1$ vector. For this reason, we will focus our attention on parallel algorithms for solving systems of linear algebraic equations.

This problem can be approached in many different ways, but all existing methods fall into one of two general categories - they are either *direct* or *iterative*. At this point, we will briefly outline the main differences between them, on the understanding that a more detailed treatment will be provided later.

Direct Methods

Two of the most commonly used techniques for solving linear equations directly are Gaussian elimination and LU factorization. Since we already discussed Gaussian elimination, we will now briefly explain how LU factorization works.

The main objective of this method is to represent matrix A as

$$A = LU \tag{17}$$

where

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \tag{18}$$

is lower triangular and

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \tag{19}$$

is upper triangular. Once such a factorization is obtained, the solution process proceeds in two stages, using the fact that

$$LUx = b \tag{20}$$

Setting

$$Ux = z \tag{21}$$

we first solve system

$$Lz = b \tag{22}$$

for z. This vector is then used to compute x as

$$Ux = z \tag{23}$$

The process of solving equation (22) is known as *forward substitution* (since it starts from the first equation), while (23) corresponds to *backward substitution* (as in Gaussian elimination)

When analyzing the properties of LU factorization, it is important to point out that matrices L and U are *not* unique. In order to avoid this problem, it is customary to set all the diagonal elements of matrix L (or U) to 1. The following example illustrates how this process works.

Example 1. Let us consider system (16) in which A and b are chosen as

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{24}$$

and

$$b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{25}$$

Our objective in the following will be to represent matrix A as

$$A = LU \tag{26}$$

where

$$L = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \tag{27}$$

and

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{28}$$

In order to ensure uniqueness, we will assume that $l_{11} = l_{22} = l_{33} = 1$.

We now proceed to compute the elements of L and U in a step-by-step manner, starting with the first row of U and the first column of L.

STEP 1. To compute the first row of U, we will make use of the fact that

$$\begin{aligned} 1 = a_{11} = l_{11}u_{11} &\implies u_{11} = 1 \\ 0 = a_{12} = l_{11}u_{12} &\implies u_{12} = 0 \\ 3 = a_{13} = l_{11}u_{13} &\implies u_{13} = 3 \end{aligned} \tag{29}$$

We can now utilize the value that we obtained for u_{11} to determine the first column of L as

$$\begin{aligned} 2 = a_{21} = l_{21}u_{11} = l_{21} \quad &\Longrightarrow \quad l_{21} = 2 \\ 1 = a_{31} = l_{31}u_{11} = l_{31} \quad &\Longrightarrow \quad l_{31} = 1 \end{aligned} \tag{30}$$

The situation after this step can be schematically represented as

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & * & 0 \\ 1 & * & * \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 0 & 3 \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix} \tag{31}$$

where $*$ denotes elements that have yet to be computed.

STEP 2. To determine the second row of U, we need l_{21}, u_{12} and u_{13} (all of which have already been calculated). Given this information, we obtain

$$\begin{aligned} 2 = a_{22} = l_{21}u_{12} + l_{22}u_{22} \quad &\Longrightarrow \quad 2 = l_{22}u_{22} \quad \Longrightarrow \quad u_{22} = 2 \\ 1 = a_{23} = l_{21}u_{13} + l_{22}u_{23} \quad &\Longrightarrow \quad -5 = l_{22}u_{23} \quad \Longrightarrow \quad u_{23} = -5 \end{aligned} \tag{32}$$

The second column of L can then be computed as

$$1 = a_{32} = l_{31}u_{12} + l_{32}u_{22} \quad \Longrightarrow \quad 1 = 2l_{32} \quad \Longrightarrow \quad l_{32} = 0.5 \tag{33}$$

using the previously calculated values of l_{31}, u_{12} and u_{22}. Once this is done, matrices L and U become

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0.5 & * \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & -5 \\ 0 & 0 & * \end{bmatrix} \tag{34}$$

STEP 3. In the final step we only need to determine u_{33}, since we know that $l_{33} = 1$. This can be done by observing that

$$1 = a_{33} = l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \quad \Longrightarrow \quad 0.5 = l_{33}u_{33} \quad \Longrightarrow \quad u_{33} = 0.5 \tag{35}$$

At this point, the factorization is complete, and matrices L and U become

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0.5 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & -5 \\ 0 & 0 & 0.5 \end{bmatrix} \tag{36}$$

We will now use the special structure of these matrices to compute vector x. As noted earlier, this procedure involves two steps, the first of which is to solve system

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{37}$$

Using forward substitution, we easily obtain

$$\begin{aligned} &z_1 = 1 \\ &2z_1 + z_2 = 0 \quad \Longrightarrow \quad z_2 = -2 \\ &z_1 + 0.5z_2 + z_3 = 1 \quad \Longrightarrow \quad z_3 = 1 \end{aligned} \tag{38}$$

Once vector z is computed, we use it to form the system

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & -5 \\ 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \tag{39}$$

which can be solved using backward substution. When we do so, we obtain

$$\begin{aligned} 0.5x_3 = 1 \quad &\Longrightarrow \quad x_3 = 2 \\ 2x_2 - 5x_3 = -2 \quad &\Longrightarrow \quad x_2 = 4 \\ x_1 + 3x_3 = 1 \quad &\Longrightarrow \quad x_1 = -5 \end{aligned} \tag{40}$$

When evaluating the efficiency of LU factorization, one should keep in mind that the number of floating point operations needed to execute it is proportional to n^3 (assuming that A is an $n \times n$ matrix, of course). This suggests that the computational effort can be prohibitive when n is large, regardless of how powerful our processors may be. Because of that, direct methods of this sort are typically applied only to *sparse* matrices, in which most of the elements are zero. We will discuss the properties of these matrices in greater detail in subsequent lectures.

Iterative Methods

Iterative methods solve system

$$Ax = b \tag{41}$$

recursively, and the vector x that they produce is only an approximation of the exact solution. To describe how these methods work, let us rewrite equation (41) as

$$Q^{-1}Ax = Q^{-1}b \tag{42}$$

where Q is an unspecified nonsingular matrix. If we now define matrix G and vector ξ as

$$G = I - Q^{-1}A \tag{43}$$

and

$$\xi = Q^{-1}b \tag{44}$$

(42) can be rewritten as

$$(I - G)x = \xi \tag{45}$$

which is equivalent to

$$x = Gx + \xi \tag{46}$$

From this, we can conclude that x^* will be a solution of equation (41) if an only if it satisfies

$$x^* = Gx^* + \xi \tag{47}$$

It can be shown that equations of this form can be solved *iteratively* as

$$x(k+1) = Gx(k) + \xi \tag{48}$$

whenever the spectral radius of matrix G satisfies $\rho(G) < 1$. If this condition is met, the process will converge the solution of (41) starting from *any* initial approximation $x(0)$.

The question that we now need to consider is whether this approach can be extended to cases where $\rho(G) \geq 1$. It turns out that this is possible if matrix Q is chosen in a particular way. The following definition and the ensuing lemma offer some insight into how this can be done.

Definition 1. The iterative sequence (48) is said to be *symmetrizable* if there exists a nonsingular matrix W such that $W(I - G)W^{-1}$ is *symmetric* and *positive definite*.

Lemma 1. If sequence (48) is symmetrizable, there exists a positive constant γ such that matrix

$$G_\gamma = \gamma G + (1 - \gamma)I \tag{49}$$

satisfies $\rho(G_\gamma) < 1$.

Remark 3. There is actually an entire range of values for γ which satisfy condition $\rho(G_\gamma) < 1$. It can be shown, however, that

$$\bar{\gamma} = \frac{2}{2 - \lambda_m(G) - \lambda_M(G)} \tag{50}$$

is the optimal choice, since it produces the smallest possible value for $\rho(G_\gamma)$.

To see how we can make use of Lemma 1, suppose that x^* is the solution of system (46) (and therefore of (41) as well). If we multiply equation (47) by γ, we obtain

$$\gamma x^* = \gamma G x^* + \gamma \xi \tag{51}$$

which becomes

$$x^* = \gamma G x^* + (1 - \gamma)x^* + \gamma \xi \tag{52}$$

after adding $(1 - \gamma)x^*$ to both sides. This implies that equation (46) and equation

$$x = \gamma G x + (1 - \gamma)x + \gamma \xi \tag{53}$$

have the *same* solution, and are therefore equivalent.

It is not difficult to see that system (52) corresponds to the iterative sequence

$$x(k + 1) = G_\gamma x(k) + \omega \tag{54}$$

where

$$G_\gamma = \gamma G + (1 - \gamma)I \tag{55}$$

and

$$\omega = \gamma \xi \tag{56}$$

Since $\rho(G_\gamma) < 1$ by virtue of Lemma 1, this sequence is guaranteed to converge to x^*.

How realistic is it to expect that an iterative sequence of the form (48) will be symmetrizable? The following lemma provides a straightforward answer to this question.

Lemma 8.2. Suppose that matrices A and Q are *symmetric* and *positive definite*. Then, there exists a nonsingular matrix W such that $W(I - G)W^{-1}$ is symmetric and positive definite as well.

It is important to recognize that this result can be applied even in cases when matrix A is *not* symmetric, because system

$$Ax = b \tag{57}$$

can always be rewritten as

$$\tilde{A}x = \tilde{b} \tag{58}$$

where

$$\tilde{A} = A^T A \tag{59}$$

and

$$\tilde{b} = A^T b \tag{60}$$

Given that is $\tilde{A}$ is symmetric and positive definite by construction (and recalling that matrix Q can be chosen freely), it follows that the conditions specified in Lemma 8.2 can be easily satisfied.

Nonlinear Algebraic Equations

Consider the system of equations

$$F(x) = 0 \tag{61}$$

where

$$F(x) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} \tag{62}$$

and functions $f_i(x_1, x_2, \ldots, x_n)$ $(i = 1, 2, \ldots, n)$ are assumed to be nonlinear and differentiable everywhere on R^n. Since our main objective is to consider the computational aspects of this problem, in the following we will assume that there exists a vector x^* such that

$$F(x^*) = 0 \tag{63}$$

It is important to keep in mind, however, that this is not a given in general, and that it is usually difficult to tell whether equations of this sort have a unique solution, multiple solutions or no solution at all. This is one of the reasons why such problems are potentially challenging.

The most effective way to solve system (61) is *Newton's method*, which starts from some initial approximation $x(0)$ and produces a sequence of iterates

$$x(k+1) = x(k) - [J(x(k))]^{-1} F(x(k)) \tag{64}$$

where

$$J(x(k)) = \begin{bmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_n \\ \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \cdots & \partial f_n/\partial x_n \end{bmatrix} \tag{65}$$

represents the Jacobian evaluated at $x(k)$.

It is not difficult to see that this approach amounts to solving a system of linear equations in each step. In order to demonstrate that, let us rewrite (64) as

$$J(x(k))[x(k+1) - x(k)] = -F(x(k)) \tag{66}$$

and set

$$A = J(x(k)) \tag{67}$$

$$y = x(k+1) - x(k) \tag{68}$$

and

$$b = -F(x(k)) \tag{69}$$

Since $J(x(k))$ and $F(x(k))$ are known after the k-th iteration, we can determine the next iterate by solving system

$$Ay = b \tag{70}$$

and computing $x(k+1)$ as

$$x(k+1) = x(k) + y \tag{71}$$

Remark 4. Since this is something that needs to be done in every iteration, the computational effort required for solving system (61) will depend on how quickly sequence (64) converges. If the number of iterations needed to achieve this happens to be N, we will need to solve N different systems of linear equations (because matrix $J(x(k))$ and vector $F(x(k))$ change in each step).

Nonlinear Differential Equations

Systems of linear algebraic equations play a key role in solving nonlinear differential equations as well. Such equations have the general form

$$\dot{x} = F(x) \tag{72}$$

where $x(t) = [x_1(t)\ x_2(t)\ \ldots\ x_2(t)]^T$ and

$$F(x) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} \tag{73}$$

As in the case of Newton's method, functions $f_i(x_1, x_2, \ldots, x_n)$ are assumed to be nonlinear and differentiable everywhere on R^n.

In order to solve such a system numerically, it is first necessary to select a *time step* h and a set of points t_0, $t_1 = t_0 + h, \ldots, t_i = t_{i-1} + h, \ldots, t_r$ for which we will compute the solution $x(t)$. Once these points have been determined, we can make use of the fact that $x(t)$ must satisfy

$$\dot{x}(t_i) = F(x(t_i)) \tag{74}$$

for $i = 0, 1, \ldots, r$.

A standard approach for computing $x(t)$ at times $\{t_0, t_1, \ldots, t_r\}$ is to approximate $\dot{x}(t_i)$ using $x(t_i)$, k previously computed values $x(t_{i-1})$, ... , $x(t_{i-k})$, and s previously computed derivatives $\dot{x}(t_{i-1})$, $\dot{x}(t_{i-2})$, ... , $\dot{x}(t_{i-s})$. To keep the notation as simple as possible, these vectors are usually denoted as $\{x_i, x_{i-1}, \ldots, x_{i-k}\}$ and $\{\dot{x}_{i-1}, \dot{x}_{i-2}, \ldots, \dot{x}_{i-s}\}$, respectively. This allows us to rewrite (74) as

$$\dot{x}_i = F(x_i) \tag{75}$$

The two most commonly used approximations for $\dot{x}_i$ are the *backward Euler formula* and the *trapezoidal formula*. The backward Euler is the simpler of the two, and approximates $\dot{x}_i$ as

$$\dot{x}_i \approx \frac{1}{h}(x_i - x_{i-1}) \tag{76}$$

The trapezoidal formula involves derivative $\dot{x}_{i-1}$ as well, and allows us to express $\dot{x}_i$ as

$$\dot{x}_i \approx \frac{2}{h}(x_i - x_{i-1}) - \dot{x}_{i-1} \tag{77}$$

To see how these approximations help us solve system (75), suppose that we are given an initial condition x_0 and that we have already computed $x_1, x_2, \ldots, x_{i-1}$ using the backward Euler method. In order to find x_i, we need to approximate $\dot{x}_i$ as

$$\dot{x}_i \approx \frac{1}{h}(x_i - x_{i-1}) \tag{78}$$

in which case equation (75) becomes

$$\frac{1}{h}(x_i - x_{i-1}) = F(x_i) \tag{79}$$

Since x_{i-1} is known at this point, (79) becomes a system of nonlinear algebraic equations in x_i, whose general form is

$$\Phi(x_i) = F(x_i) - \frac{1}{h}x_i + \frac{1}{h}x_{i-1} = 0 \tag{80}$$

This is obviously something that we can solve using Newton's method, so we once again have a situation where the main computational task is to solve a system of linear equations of the form

$$Ax = b \tag{81}$$

Note, however, that in the case of differential equations we will have to do this for *every point* t_i, so accellerating the process using parallel techniques becomes even more critical.

The trapezoidal formula has similar computational requirements, since it approximates system (75) as

$$\frac{2}{h}(x_i - x_{i-1}) - \dot{x}_{i-1} = F(x_i) \tag{82}$$

If we rewrite this expression as

$$\Phi(x_i) = F(x_i) - \frac{2}{h}x_i + \left[\frac{2}{h}x_{i-1} + \dot{x}_{i-1}\right] = 0 \tag{83}$$

we obtain a system of nonlinear algebraic equations in x_i, which can be solved using Newton's method.

Remark 5. Note that the term

$$b = \left[\frac{2}{h}x_{i-1} + \dot{x}_{i-1}\right] \tag{84}$$

is treated as a constant in equation (83), because x_{i-1} and $\dot{x}_{i-1}$ were already computed in the previous step (and are therefore known at this point).