Design and implementation of parallel systems using unparallel programming languages

Bahaa mohsen zbeel
College of fine art
Bahaamohsen95@yahoo.com

Abstract
This paper describes a technique of simplifying programming systems that exhibits parallelism in its design using traditional (unparallel) programming languages instructions in Pentium -windows platform based serial computers. A brief overview of parallel programming and parallel computing theories and concepts is presented. The research use well known parallel computing model such as artificial neural net and Petri nets as demonstrations examples for simplifying its programming using Visual basic 6 as unparallel programming language.

Keyword: parallel programming, parallel computing, serial computer.

1. Introduction
Even as processor speeds and memory capacities continue to rise, they never seem to be able to satisfy the increasing demands of software; scientists and engineers routinely push the limits of computing systems to tackle larger, more detailed problems. In fact, they often must scale back the size and accuracy of their simulations so that they may complete in a reasonable amount of time of fit into memory. The amount of processing power directly impacts the scale of the problem being addressed [Hahn Kim 2003].

One option for improving performance is parallel processing. There have been decades of research and development in hardware and software technologies to allow programs to run on multiple processors, improving the performance of computationally intensive applications. Yet writing accurate, efficient, high performance parallel code is still highly non-trivial, requiring a substantial amount of time and energy to master the discipline of high performance computing, commonly referred to as HPC [Hahn Kim 2003]. As multi-core processors are becoming ubiquitous, parallel programming is the technology that can make them succeed or not. Parallel programming improves performance by breaking down a problem into smaller sub problems that are distributed to multiple processors. Thus, the benefits are two-fold. First, the total amount of computation performed by each individual processor is reduced, resulting in faster computation. Second, the size of the problem can be increased by using more memory available on multiple processors [Hahn Kim 2003]. The proposed language for building parallel program is visual basic 6. The most important facility in visual basic 6 that used for simplify system programming that exhibits parallelism is (timer control) that enables the programmer to insert code in its application and run it independently from the other part of the application.
The rest of the paper is organized as follows: Section 2 provides a brief overview of parallel programming theories and concepts. Section 3 discusses the detailed implementation of simplify parallel system programming in Visual Basic 6. Section 4 presents conclusion and future works.

2. Parallel programming

Innovations in hardware architecture, like hyper threading or multi-core processors, make parallel computing resources available for inexpensive desktop computers. However, the use of these innovations requires parallel programming techniques. In a few years, many standard software products will be based on concepts of parallel programming to use the hardware resources of future multi-core processors efficiently. Thus, the need for parallel programming will extend to all areas of software development. The application area will be much larger than the area of scientific computing, which used to be the main area for parallel computing for many years. The expansion of the application area for parallel computing will lead to an enormous need for software developers with parallel programming skills. Some chip manufacturers already demand to include parallel programming as a standard course in computer science curricula. [Thomas Rauber 2010]

2.1 Amdahl’s Law

Amdahl’s Law is a useful principle for determining the theoretical maximum speedup a parallel program can achieve. As stated in [D. Patterson 2003], “Amdahl’s Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.” In our case, the “faster mode of execution” is parallelizing one or more sections of code. Amdahl’s Law can be concisely described using the following equation, where $f$ is the fraction of time spent on serial operations and $p$ is the number of processors: [Hahn Kim 2003]:

$$speedup \leq \frac{1}{f + \frac{(1 - f)}{p}}$$

From this equation, we see that as the number of processors increases, the term $\frac{(1-f)}{p}$ approaches 0, resulting in the following:

$$\lim_{p \to \infty} speedup \leq \frac{1}{f}$$

As we can see, the maximum possible speedup for a parallel program depends on how much of a program can be parallelized. Consider three programs with three different values of $f$: 0.1, 0.01, and 0.001. The maximum possible speedups for these programs are 10, 100, and 100, respectively. Figure 1 shows the speedups for these programs up to 1024 processors: [Hahn Kim 2003].
Amdahl’s Law clearly demonstrates the law of diminishing returns: as the number of processors increases, the amount of speedup attained by adding more processors decreases [Hahn Kim 2003].

Amdahl’s Law was originally defined for processor architectures, but applies equally well to parallel programming. Suppose 50% of a program’s runtime can be parallelized. Amdahl’s Law states that even if the runtime of the parallelized portion can be reduced to zero, total runtime is only reduced by 50%, as shown in Figure 2. Thus, Amdahl’s Law computes the theoretical maximum performance improvement for parallel programs [Hahn Kim 2003].

2.2 Parallel Computers Architectures

For quite a while now, the top computers have been some sort of parallel computer, that is, an architecture that allows the simultaneous execution of multiple instructions or instruction sequences. One way of characterizing the various forms this
can take is due to Flynn [Victor 2011]. Flynn’s taxonomy distinguishes between whether one or more different instructions are executed simultaneously, and between whether that happens on one or more data items. The following four type's result, which we will discuss in more detail below[Hahn Kim 2003]:

**SISD** Single Instruction Single Data: this is the traditional CPU architecture: at any one time only a single instruction is executed, operating on a single data item.

**SIMD** Single Instruction Multiple Data: in this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item. Vector computers are typically also characterized as SIMD.

**MISD** Multiple Instruction Single Data. No architectures answering to this description exist; one could argue that redundant computations for safety-critical applications are an example of MISD.

**MIMD** Multiple Instruction Multiple Data: here multiple CPUs operate on multiple data items, each executing independent instructions. Most current parallel computers are of this type. Table 1 demonstrates the architecture of parallel computer.

<table>
<thead>
<tr>
<th></th>
<th>Single data</th>
<th>Multiple data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single instruction</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple instruction</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

### 2.3 parallel programming models

Classifications of parallel programming models can be divided broadly into two areas: process interaction and problem decomposition. [Singh 1997]

**2.3.1 Process interaction**

Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing, but it can also be implicit.

**I-Shared memory**

The shared memory model is based on a set of threads that is created when parallel operations are executed. This type of computation is also called *fork-join parallelism*. Threads share a global address space and thus access array elements via a global index. The main parallel operations are *parallel loops* and *parallel sections*. Parallel loops are executed by a set of threads also called *a team*. The iterations are distributed among the threads according to a predefined strategy. This scheduling strategy implements the chosen domain decomposition. Parallel sections are also executed by a team of threads but the tasks assigned to the threads implement different operations. This feature can for example be applied if domain decomposition itself does not generate enough parallelism and whole operations can be executed in parallel since they access different data structures.

**II-Message passing**

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. It is a natural fit for programming distributed memory machines but also can be used on shared memory computers. The domain decomposition is implemented by developing a code describing the local computations and local data structures of a single process. Thus, global arrays have to be split up and only the local part has to be allocated in a process. This handling of global data structures is called *data distribution*. Computations on the global arrays also have to be
transformed, e.g., by adapting the loop bounds, to ensure that only local array elements are computed. Access to remote elements has to be implemented via explicit communication, temporary variables have to be allocated, messages have to be constructed and transmitted to the target process. [Bernd Mohr 2006]

III- Implicit
In an implicit model, no process interaction is visible to the programmer, instead the compiler and/or runtime is responsible for performing it. This is most common with domain-specific languages where the concurrency within a problem can be more prescribed. [Singh 1997]

2.3.2 Problem decomposition
Any parallel program is composed of simultaneously executing processes, problem decomposition relates to the way in which these processes are formulated. This classification may also be referred to as algorithmic skeletons or parallel programming paradigms.

I-task parallelism
In the compute-centric view of a problem, we focus on defining a set of tasks that operates on data. We should use this pattern when the problem is conveniently separated into tasks that can execute concurrently and the distribution of data working set is of secondary concern for performance. This assumption is often valid for systems with a shared memory space, where concurrently executing tasks have small working sets and infrequent access to shared global data structures. [Timothy 2004]

II data parallelism
In the case where distribution of data is crucial for execution efficiency, we should use the Data Parallel Algorithm Strategy Pattern, where the problem is expressed in terms of a single stream of tasks applied to each independent element of a data structure, and the solution involves efficient execution of tasks with data distribution as a primary concern. [Timothy 2004]

2.4 Parallel Programming Goals
The three major goals of parallel programming (over and above those of sequential programming) are [Paul 2011]:
1-Performance.
2- Productivity.
3- Generality.

2.4.1 Performance
The key point to understand is that parallel programming is primarily a performance optimization, and, as such, it is one potential optimization of many. If your program is fast enough as currently written, there is no reason to optimize, either by parallelizing it or by applying any of a number of potential sequential optimizations.

By the same token, if you are looking to apply parallelism as an optimization to a sequential program, then you will need to compare parallel algorithms to the best sequential algorithms. This may require some care, as far too many publications ignore the sequential case when analyzing the performance of Parallel algorithms.

2.4.2 Productivity
One of the inescapable consequences of the rapid decrease in the cost of hardware is that software productivity grows increasingly important. It is no longer sufficient merely to make efficient use of the hardware, it is now also necessary to make extremely efficient use of software developers. This has long been the case for sequential hardware, but only recently has parallel hardware become a low-cost
commodity. Therefore, the need for high productivity in creating parallel software has only recently become hugely important.

2.4.3 Generality

One way to justify the high cost of developing parallel software is to strive for maximal generality. All else being equal, the cost of a more-general software artifact can be spread over more users than can a less-general artifact. Unfortunately, generality often comes at the cost of performance, productivity, or both. The nirvana of parallel programming environments, one that offers world-class performance, productivity, and generality, simply does not yet exist. Until such a nirvana appears, it will be necessary to make engineering tradeoffs among performance, productivity, and generality.

2.5 Parallel languages

One approach to mitigating the difficulty of parallel programming is the design of languages that offer explicit support for parallelism. There are several approaches, and we will see some examples [Victor 2011]

- Some languages reflect the fact that many operations in scientific computing are data parallel. Languages such as High Performance FORTRAN (HPF) have an array syntax, where operations such addition of arrays can be expressed \( A = B+C \). This syntax simplifies programming, but more importantly, it specifies operations at an abstract level, so that a lower level can make specific decision about how to handle parallelism. However, the data parallelism expressed in HPF is only of the simplest sort, where the data are contained in regular arrays. Irregular data parallelism is harder; the Chapel language makes an attempt at addressing this.

- Another concept in parallel languages, not necessarily orthogonal to the previous, is that of Partitioned Global Address Space (PGAS) model: there is only one address space (unlike in the MPI model), but this address space is partitioned, and each partition has affinity with a thread or process. Thus, this model encompasses both Symmetric Multi Processing SMP (the programming model for architectures on Uniform Memory Access) and distributed shared memory. A typical PGAS language, Unified Parallel C (UPC), allows you to write programs that for the most part look like regular C code. However, by indicating how the major arrays are distributed over processors, the program can be executed in parallel[Victor 2011].

3. The detailed implementation

In this section the detailed implementation using visual basic 6 programming language are discussed. The goal is to obtain simplify system building that exhibit parallelism in its nature by eliminate the control structure needed such as loops and conditions needed to build such application in unparallel programming language. The main tool used in visual basic 6 is 'timer control'.

The purpose of the Timer control is to add a link to the system timer. It will fire an event at an interval you specify. Once you place a timer on the form, you can set the interval for it by using its properties. The Timer object has one event: the Timer event that fires whenever the specified interval elapses [Root 2006]. The Timer control will not display to the user at runtime. It’s a hidden object during Run mode, so you will only see it in Design mode [Root 2006].

The length of time, measured in milliseconds, is set with the Interval property to be any number from 0 to 65,535 (about 1 minute and 5 seconds). The event triggered each time Timer1.Interval milliseconds elapses is called Timer1_Timer ( ). In order to begin timing, a timer must first be turned on by setting its Enabled property to True. A timer is turned off either by setting its Enabled property to False or by setting its Interval property to 0.
3.1 case studies
In this section two case studies are considered, perceptron neural network and a simple communication protocol Petri net model to demonstrate the effectiveness of redesigning them in parallel like programming in visual basic.

3.1.1 Case study 1: perceptron neural network
The perceptron is the simplest form of a neural network used for the classification of patterns said to be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane). Basically, it consists of a single neuron with adjustable synaptic weights and bias. The perceptron built around a single neuron is limited to performing pattern classification with only two classes (hypotheses). By expanding the output (computation) layer of the perceptron to include more than one neuron, we may correspondingly perform classification with more than two classes. However, the classes have to be linearly separable for the perceptron to work properly [Simon 1999].

In this work, two input layer perceptron with one output neuron layer to solve (bitwise and) problem shown in figure 3, this perceptron is used for its simplicity to focus on the parallelism aspect of the research.

Fig. 3: Two-Input/Single-Output Perceptron

P: represent the input vector.
W: represent the weights vector.
b: the bias values.
a: the output of the perceptron according to hard limit function
The hard limit function is the transfer function defined as:
\[ a = \text{hardlim}(Wp + b) \]

Where \( n = w_{1,1}p_1 + w_{1,2}p_2 + b \).

The learning rule for the perceptron is:
\[ w_{\text{new}} = w_{\text{old}} + \mu (d - a)p \]
Where: \( \mu \): represent learning rate, \( d \): represent the desired output, in contrast with 'a' that represent the actual output.

The Perceptron Algorithm
1. Initialize the weights (either to zero or to a small random value)
2. Pick a learning rate \( \mu \) (this is a number between 0 and 1)
3. Until stopping condition is satisfied (e.g. weights don't change):
   3.1 For each training pattern (p, d):
      3.1.1 Compute output activation \( a = f (W, p) \)
      3.1.2 If \( a \neq d \) update the weights:
3.1.2.1 \( w_{\text{new}} = w_{\text{old}} + \mu (d-a)p \)

4. End

3.1.2.2 \( b_{\text{new}} = b_{\text{old}} + \mu (d-a) \)

In case of (bitwise And), the value of \( \mu \) assumes to be 1 for simplicity, \( b=-3 \) and the resulted weights as follows: \( w_{1,1}=2, \quad w_{1,2}=2 \).

The patterns and the desired output obtained from the and truth table in table 2.

Table 2: And truth table

<table>
<thead>
<tr>
<th>Pattern no.</th>
<th>P1</th>
<th>P2</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pattern 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Pattern 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pattern 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Two operations in this perceptron neural net are independent, in learning stage in each iteration of learning algorithm, there is:

\[
W_{1,1}^{\text{new}} = W_{1,1}^{\text{old}} + \mu (d_1-a_1)P_1,
\]

\[
W_{1,2}^{\text{new}} = W_{1,2}^{\text{old}} + \mu (d_1-a_1)P_2.
\]

And in the testing stage, on input layer, the following operations are independent:

\( W_{1,1} \cdot P_1, W_{1,2} \cdot P_2 \)

So the ability of simplifying the programming of the perceptron algorithm exists in Visual Basic 6 using the timer control. The following Visual Basic form in figure 4 demonstrates the idea:

The controltimer timer controls the number of iteration of the net. For simplicity the program use the number of patterns as termination criteria. This timer also turns on the input layer to be start. 'Controltimer' its name in the program. The two timer's neuron1 and neuron2 in the program refers to the units of input layer. Each one weight the input by multiply it by certain value using scalar variable p1 and p2 as input pattern( operand of bitwise And), the scalar variable weight1 and weight2 as there weights respectively. The two timers weightupdate1 and weightupdate2 perform the weights update task. And the timer outputneuron perform the hard limit transfer function task of the perceptron. In this design the parallelism occur at input layer in testing stage and in performing weight updating in training stage. The transfer
function of the perceptron neural net is serial task with input layer neurons operations and weight update in training stage. Input layer neurons operations and weights update are serial with each other.

This design eliminates the need of loops and matrices in traditional programming for the above perceptron learning algorithm. The following code is for each input layer neurons (timers).

For input neuron 1:
- 'p1: represent input for input neuron 1
- p1weight1 = p1 * weight1 ' weight the input
- neuron1.Enabled = False ' deactivate the neuron after performing its task

For input neuron 2:
- 'p2: represent input for input neuron 2
- p2weight2 = p2 * weight2 ' weight the input
- neuron2.Enabled = False ' deactivate the neuron after performing its task

For weight updating to the neuron 1:
- If actual_output <> desired_output Then
  - weight1 = weight1 + m * (desired_output – actual_output) * p1
  - bias = bias + m * (desired_output – actual_output)
- End If
- weightupdate1.Enabled = False ' deactivate this task

for weight updating to the neuron 2:
- If actual_output<> desired_output Then
  - weight2 = weight2 + m * (desired_output – actual_output) * p2
- End If
- weightupdate2.Enabled = False ' deactivate this task

3.1.2 Case study 2: A simplified Petri net model for communication protocol.

Petri nets are graphical and mathematical modeling tool applicable to many systems. They are promising tool for describing and studying information processing systems that are characterized as being concurrent, and/or stochastic. As graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. A Petri net is a particular kind of directed graph, together with an initial state called the initial marking, m0, the underlying graph N of a Petri net is a directed, weighted, bipartite graph consisting of two kinds of nodes, called places and transitions. In modeling, using the concept of conditions, and transitions represent events. A transition (an event) has a certain number of input and output places representing the pre-conditions and post-conditions of the events respectively, the presence of the token in a place is interpreted as holding the truth of the conditions associated with the place, in another interpretation, k tokens are put in a place to indicate that k data items or resources are available. Communication protocols are another area where Petri nets can be used to represent and specify essential features of a system. The liveness (deadlock free operation) and safeness (no overflow in memory represented by places) properties of a Petri net are often used as correctness criteria in communication protocols. The Petri net shown in figure 5 is a very simple model of a communication protocol between two processes [T. Murata 1989].
Figure 5: A simplified model of a communication protocol.

The visual basic design proposed for Petri net model of the above communication protocol is depicted in figure 6. In this design, transitions represented by timers (that runs independently and in parallel) and places (input and output places) represented by text boxes (or any variable or visual basic object). The timers (transitions) checks the value (the truth of its firing condition) in textboxes (its input places that holds its firing or activation or enabling conditions) to produce another values (results) in another textboxes (output places that holds conditions firing for another transitions) that checks by another timers.

Figure 6: A Petri net communication protocol model using timer control in visual basic 6.

In this design, the liveness of a certain transition for a given marking (tokens in the places) for example, can be tested by inserting number of numeric values in the textboxes (tokens in the places) and wait for a transition to be fired, if not then the transition with respect to the given marking is dead. The program enabled checkbox control for each fired transition the following visual basic code is for process1 transition in the communication protocol Petri net model listed as a demonstration example.

Private Sub process1transition_Timer()
    temp = ackreceivedplace.Text

1886
inputtokens = Val(temp)
If inputtokens > 0 Then
inputtokens = inputtokens - weight
ackreceivedplace.Text = Str(inputtokens)
temp = readytosendplace.Text
outptokens = Val(temp)
temp = Str(weight + outptokens)
readytosendplace.Text = temp
process1transition.Enabled = False
pocess1flag.Enabled = True
pocess1flag.Value = rising
End If
End Sub

The programming simplification in the above design occurs because the liveness of a transition under study for a given marking can be tested, without needs of programming an algorithms use an incidence matrix or reachability graph or an algorithms that test certain characteristics in the net to decide the liveness of a transition, all what's we need is to redesign the Petri net as illustrated above and runs it for a given marking and finally checks if there is any disabled checkboxes (represent the transitions firing indicator) left, so these transitions are dead for that marking. Else the net is live.

The above implementation of Petri net in visual basic using timer control is optimal because all what's need the programmer is to replace transitions by timer controls and places by text boxes or any other data controls or simply variables and run the program to discover directly the properties of the modeled system.

4. Conclusions and future works

The main conclusions that can be extracted from this research are:

1. the use of unparallel programming language instructions that can be executed independently from other parts of the program simplify the programming task of these system and increase programming productivity.
2. In general, the benefit gained from the proposed technique in this paper increase significantly as similar parallel unit in the system increase because its reduce the time needed to program these units. All the effort needed is to replicate the code needed to each parallel unit run in the parallel system.
3. in artificial neural networks, the gain from the proposed technique maximized when the neurons increased and serial parts decreased (such as successive layers in feed forward neural networks that each layer can't run unless its previous layer run ). The type of neural network affects the benefit gained according to parallelism nature included in it, as example, parallel implementation of Hopfield neural net where each neuron activate in parallel with each other is perfect application of the technique proposed in this paper because parallel Hopfield is one layer neural net contains neurons run independently. in the other hand, back propagation neural net contains layers run in successive manner and this reduce the benefit gained from the proposed technique because its need an effort to control the execution order of each other such as in case study 1.
4. for Petri net, the proposed technique is perfect because Petri net is the main theoretical tool to describe the parallelism nature in systems, so the proposed technique is one to one mapping for any Petri net. the properties of the system can be
tested directly without need for specialized algorithm such as the liveness property of simple communication protocol in case study 2.

The future work that can be done is to use threading techniques to programming systems that exhibits parallelism to simplify programming task and speeding up it.

References
Paul E. McKenney, January 5, 2011" Is Parallel Programming Hard, And, If So, What Can You Do About It?",Linux Technology Center IBMBeaverton. paulmck@linux.vnet.ibm.com
Root, Randal; Romero Sweeney, Mary ,2006" A tester's guide to .NET programming", Apress. p. 3.
Simon haykin,1999"neural network acomprehensive foundation" second edition,printice hall international,inc.