A Practical Application of FGDLS to Birds Flock Trajectory

Sabin Tabirca Tatiana Tabirca
University College Cork, Department of Computer Science, BCRI
College Road, Cork, Ireland

Lawrence Tianruo Yang
St. Francis Xavier University, Department of Computer Science
P.O.Box 5000, Antigonish, B2G 2W5, Canada

Andrea Unger Thomas Steube
Otto-von-Guericke-University of Magdeburg, Department of Computer Science
Magdeburg, Germany

Abstract: - In this paper we describe a practical application of the Feedback Guided Dynamic Loop Scheduling (FGDLS) algorithm. FGDLS is a recent scheduling method that was proposed in Bull [1] to deal with a sequence of similar or identical parallel loops. The presumption is that the parallel loops are very similar with the same number of iterations and with iterations that do not vary much from one step to another. The FGDLS algorithm uses feedback information from the previous parallel iteration e.g. measured execution times to schedule the current parallel loop. So far all the applications of FGDLS have considered only identical iterations within the parallel loops. In this article we will propose a practical application to simulate the flocking birds (boids) trajectory. For this application the iterations are not the same varying slightly from one parallel loop to another.

Key-Words: Dynamic scheduling, boid algorithm, visualization.

1 Introduction

Load imbalance is the most important overhead in many parallel applications. Because loop structures represent the main source of parallelism, the scheduling of parallel loop iterations to processors can have a significant effect on minimizing load imbalance. Among many algorithms for loop scheduling, Feedback Guided Dynamic Loop Scheduling (FGDLS) is one that has been suggested recently. The FGDLS algorithm was introduced by Bull [2], and successively developed by Tabirca et.al. [10] and [11]. The FGDLS method was applied to solve several practical and theoretical applications from Numerical Weather Prediction, Linear Algebra or Graph Theory (see [1], [12] or [13]).

2 FGDLS Method

In this section some properties of the FGDLS method are reviewed. More information about the method can be found in [2], [3], [10] and [11]. We assume that the loop structure of Figure 1 is to be scheduled on a parallel machine (usually shared memory) with p processors $P_1, P_2, \ldots, P_p$. We also assume that the workload associated with the routine `loop_body(i)` at the outer parallel loop t is given by $\{w_i, i = 1, 2, \ldots, n\}$ and more importantly this does not vary much from a parallel loop

```
do seq t = 1, tmax
    do par i = 1, n
        call loop_body(i)
    end do
end do
```

Figure 1: The FGDLS loop structure

to another. The FGDLS method uses block partitioning of the parallel iterations on the processors which can be given by l_j^t and h_j^t, the lower and upper bounds of the iteration block assigned to the processor j at the time t. These bounds satisfy the equation

$$l_1^t = 1,\ l_{j+1}^t = h_j^t + 1, j = 1, \ldots, p-1,\ h_p^t = n. \quad (1)$$

FGDLS starts with the initial, arbitrarily chosen, loop bounds $(l_j^1, h_j^1)_{1 \le j \le p}$.
New loop bounds $(l_j^{t+1}, h_j^{t+1})_{1 \le j \le p}$ are calculated from the bounds $(l_j^t, h_j^t)_{1 \le j \le p}$ by approximately equally distributing the total workload onto the processors. We assume that the execution time of each processor at time t (the accumulated execution time for `loop_body(i)`, $i = l_j^t, l_j^t + 1, \ldots, h_j^t$) is given by $T_j^t = \sum_{i=l_j^t}^{h_j^t} w_i,\ j = 1, 2, \ldots, p$. A piecewise constant approximation of the workload at time t is then given by

$$w_i^t = \frac{T_j^t}{h_j^t - l_j^t + 1} = \frac{\sum_{i=l_j^t}^{h_j^t} w_i}{h_j^t - l_j^t + 1},\ \ l_j^t \le i \le h_j^t. \quad (2)$$

New loop bounds, $l_j^{t+1}, h_j^{t+1}, j = 1, 2, \ldots, p$, are calculated so that this piecewise constant workload $\{w_i^t, i = 1, 2, \ldots, n\}$ is approximately equally distributed over the p processors:

$$\sum_{i=l_j^{t+1}}^{h_j^{t+1}} w_i^t \simeq \frac{1}{p} \cdot \sum_{i=1}^{n} w_i^t = \overline{W}. \quad (3)$$

A simple way to solve this approximate distribution problem is to calculate the loop upper bounds as follows

$$h_j^{t+1} = h \quad \Leftrightarrow \quad \sum_{i=l_j^{t+1}}^{h} w_i^t \le \overline{W} < \sum_{i=l_j^{t+1}}^{h+1} w_i^t. \quad (4)$$

The lower bounds are then given by

$$l_{j+1}^{t+1} = h_j^{t+1} + 1,\ j = 1, 2, \ldots, p-1.$$

Equation (4) can generate an $O(p + \log p)$ scheduling algorithm (see Tabirca et.al. [**?**]).

3 The Boid Trajectory Problem

Several research studies and books for birds flocking trajectory have been written over the last 20 years but all of are based on the flocking behavior model introduced by Craig Reynolds [8] and [9]. This model which is called "Boid Behavioral Model" can be also applied to a variety of flocking from ants and fishes to people. It provides autonomous behavior for every element of the flock by using a relatively easy mathematical model.

Three basic rules are applied for every bird: alignment, cohesion and separation (see Figure 2). Alignment is related to the fact that birds (or swarms and crowds in general) head to the same direction. So the alignment of a bird is influenced by the average alignment of its neighbors. Cohesion and separation are set against each other. On the one hand, the flock is one unity that should not be separated. The birds need to stay close. The boid model takes this into account by heading each bird to the average position of its flock neighbors. As a result the bird stays within the flock. Cohesion leads to closer distances between the birds until they clash. To avoid this, the counterpart separation is used. Every bird steers away from its closest flock mates to keep a comfortable space that allows undisturbed flight. These three forces are applied to the current velocity, which contains heading and speed. The new position of a bird is computed based on old position and resulting velocity as well as the data of the other birds in the flock.

Let's assume that the bird flock has n birds. As every bird is influenced by its neighbors, the calculation for each bird needs to consider data about its flock mates. This means that the new position of a bird must consider n data (position and velocity) of the whole flock as Figure 3 illustrate. This results in a time-consuming computation for a large number of birds as the positions are

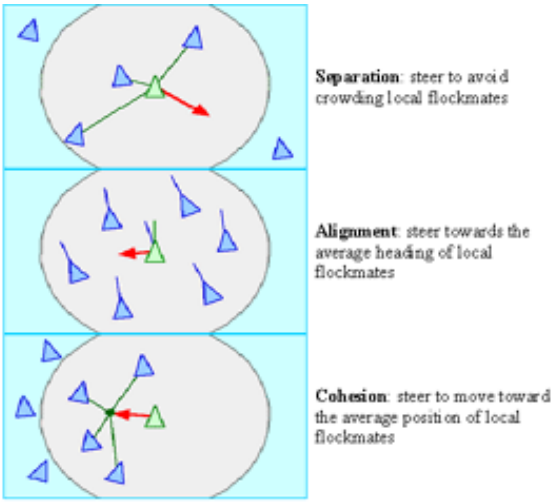

Figure 2: Rules for Steering Behaviours.

found in $O(n^2)$. Moreover, parallel computation can be employed to calculate the new positions of the boid.

---

do seq $t = 1, tmax$
  do par $i = 1, n$
    call position(i, $List_i^t$);
  end do
end do

Figure 3: The Main Loop Calculation for Boids.

A widely accepted way to reduce the complexity [9] is to keep for each bird a list with all the birds in its close vecinity at that moment

$$List_i^t = \{j : \ dist(bird_i, bird_j) < d\}, \ i = 1, ..., n.$$

The new position of the bird i considers in this case only the birds from $List_i^t$ which are in the vecinity. The main advantage of this approach is the computation of the new position, which is done by the call of position(i, $List_i^t$) does not use the whole n data but fewer. So that the complexity of the computation decreases from $O(n^2)$ to $O\left(\sum_{i=1}^{n} |List_i^t|\right)$.

One can see that the call depends of time as the list $List_i^t$ can vary from one iteration to another. However, the variation of the list $List_i^t$ over time is slow so that we can presume that the numbers of birds in two consecutive lists $List_i^t$ and $List_i^{t+1}$ are the same or vary only slightly. So the FGDLS method can be used to schedule the loop structure.

4 FGDLS Solution for the Boid Trajectory Problem

In this section describes how FGDLS can be applied to solve the Boid trajectory problem. Figure 4 gives the pseudo-code of this solution that simulates the trajectory of a boid. Initially, it should be a preprocessing to initialize with random values the data (coordinates, velocity, etc) for each bird. This is followed by another preprocessing to calculate the lists $List_j^0, \ j = 1, ..., n$.

---

procedure boid(n)
// generate initial data (x,y,vel)
  do par $i = 1, n$
    generate($data_i$);
  end do
// generate initial lists
  do par $i = 1, n$
    generate(i,$data$,$List_i$);
  end do
// generate initial bounds
  do par $j = 0, p$
    $h_j^0 = j * n/p$;
  end do
// apply FGDLS
  do seq $t = 0, tmax$
    // Compute the lower bounds
    do par $j = 1, p$
      $l_j^t = h_{j-1}^t + 1$;
    end do
    do par $j = 1, p$
      //Iterations assigned to Processor j
      do seq $i = l_j^t, h_j^t$
        call position(i, $List_i$,$data_i$);
      end do
      Compute the execution time T_j^t;
    end do
    call fgdls(h^t, T^t, h^{t+1});
    end if
  end do
end.

Figure 4: FGDLS Computation for Boids.

The FGDLS method starts from the initial up-

per bounds $h_j^0 = j * n/p, j = 1, ..., p$. From the upper bounds, the lower bounds $l_j^t = h_{j-1}^t, j = 1, ..., p$ are computed at any step of the simulation t. The parallel loop of Figure 3 can be now scheduled on processors so that each processor P_j receives the iterations $i = l_j^t, l_j^t + 1, ..., h_j^t$. After this parallel computation the execution times $T_j^t, j = 1, ..., p$ are observed. Based on the feedback information given by the upper bounds $h_j^t,\ j = 1, ..., p$ and the execution times $T_j^t, j = 1, ..., p$ the call of fgdls(h^t, T^t, h^{t+1}) finds the upper bounds $h_j^{t+1},\ j = 1, ..., p$ for the step $t + 1$.

Theoretical studies ([3] and [10]) have shown that FGDLS achieves a very good load balance. This means the execution times $T_j^t, j = 1, ..., p$ are the same after the first few steps t. When the workloads of the loop iterations do not vary from one step to another, the upper bounds become stable $h_j^t = h_j^{t+1},\ j = 1, ..., p$ so that each processor receives the same iterations. If this stability is detected then there is no point in calculating the new bounds as the previous bounds can be used. In this way FGDLS achieves a perfect load balance for almost all the parallel loops and keeps the scheduling overhead minimal as only the few sequences of upper bounds are usually computed. However, since the workloads vary slightly we do not expect to achieve stability and the upper bounds must be calculated at any time.

5 Practical Experiments

The boid trajectory problem was implemented on a Silicon Graphics Origin 2000 parallel computer with 64 R10000 processors (running at 195 MHz). The implementation used the following scheduling algorithms: block scheduling (B), self-scheduling (SS) and feedback guided dynamic loop scheduling (FS). The block scheduling algorithm divides the parallel loop of Figure 3 into p equal-sized blocks of iterations. This schedule can lead to load imbalance, although the computation cost of the scheduling algorithm is small, and there are no synchronization overheads. However, the scheduling can give good results when the lists $List_i,\ i = 1, ..., n$ have a similar number of elements.

The self-scheduling algorithms keep a central queue with the parallel iteration indices. When a processor finishes the computation associated with a given iteration index, it removes the next loop iteration index from the queue and executes the associated computation. The simple block self-scheduling algorithm [4] allows a processor to take a chunk of k iterations from the queue; this reduces the contention of multiple processors accessing the shared queue, but can lead to some load imbalance. To avoid this poorer load balance, guided self-scheduling algorithms, see [5], dynamically change the chunk size, starting from a large size and making it progressively smaller. The way in which the chunk size decreases gives rise to several different guided self-scheduling algorithms: adaptive guided self-scheduling [6], factoring [7] and trapezoid self-scheduling [14]. Trapezoid self-scheduling was used for also used to schedule the parallel loops from Figure 3 as it is recognized to be the most efficient of this class.

The Continuous FGDLS algorithm was used to implement the call of fgdls(h^t, T^t, h^{t+1}). We know that algorithm has a very low scheduling overhead and achieves an optimal load balance in several cases [10], [11]. The implementation of this scheduling follows exactly the pseudo-code illustrated in procedure boid(n).

The computation was done using a boid of 5000 birds whose trajectory was simulated 10000 times. Usually, the simulation is associated with visualization too. An appropriate visualization requires optimally 25 frames per second. The the number of frames is smaller than 10 the effect of standstill becomes unavoidable. With more and more birds animated, the computation slows down and so does the visualization if the scene is drawn once after each computation. A sequential computation for this number of birds can only simulate 8 frames per seconds which is not acceptable. In the following some experimental results are presented when $p = 4$ processors are used. The initial distribution of birds is done so that most of the birds are grouped into a central flock with the rest scattered around it. After a few steps of simulation all the birds are gathered into the central flock.

The first experiment follows the execution times per processors for these three methods. They are presented below in milliseconds and reflect the computation for $t = 1, 2, 3, 4, 5$. For the uniform block

scheduling the execution times are not balanced and they are going to be like that for most of the computation steps. The execution times for the trapezoid self-scheduling are quite well balanced started from the first iteration and stay well balanced for the whole execution. For the FGDLS method the execution times per processor are not balanced initially but the method reaches the load balance in only 4 steps. An important measure is the average of those execution times which is 32.45 ms for SS and 31.5 ms for FS. The difference of 1 ms quantifies the scheduling overhead that is bigger for SS. Certainly, for the whole 10000 iterations this difference cumulates in more than 10 second which is a big gap.

```
Uniform Block Scheduling
-----------------------------
t=1    28.5  34.2  32.7  33.1
t=2    26.3  35.5  33.0  35.4
t=3    27.4  33.1  34.8  34.5
t=4    27.8  32.2  33,9  32.6
t=5    29.2  33.9  34.6  33.4
Trapezoid Self-Scheduling
-----------------------------
t=1    32.3  32.7  32.8  32.0
t=2    32.5  32.2  32.4  32.9
t=3    32.9  32.8  32.2  32.5
t=4    32.1  32.6  32.3  32.9
t=5    32.8  32.4  32.3  32.2
FGDLS
-----------------------------
t=1    28.6  34.1  32.3  33.0
t=2    31.5  33.2  32.6  31.3
t=3    31.9  31.0  31.5  31.2
t=4    31.4  31.5  31.6  31.4
t=5    31.4  31.5  31.4  31.6
```

The second experiment shows the overall execution times for the whole simulation when the number of processor varies. These execution times expressed in seconds are presented in Table 1 and Figure 5. They show that the execution times for the trapezoid self-scheduling and FGDLS scheduling are nearly the same but with a small advantage for the latter.

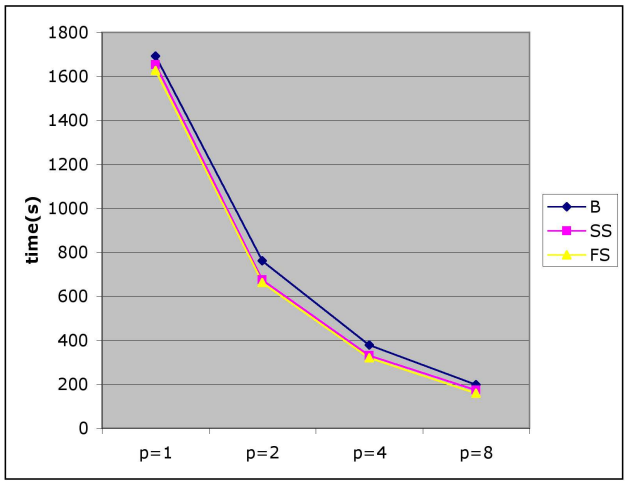

Figure 5: Execution Times for $p = 1, 2, 4, 8$.

6 Final Conclusions

This article has presented an application of the FGDLS scheduling to simulate the trajectory of bird flocks. This is a practical application of the method where the workloads of parallel iterations vary slightly. The method has been tested against trapezoid self-scheduling and uniform block scheduling. Practical experiments have shown that FGDLS offers the smallest execution times.

Acknowledgments: Research supported by the Boole Centre for Research in Informatics at UCC.

References:

[1] J.M.Bull, R.W.Ford and A.Dickinson (1996) A Feedback Based Load Balance Algorithm for Physics Routines in NWP, Proceedings of Seventh Workshop on the Use of Parallel Processors in Meteorology, World Scientific.

[2] J.M.Bull (1998) Feedback Guided Loop Scheduling: Algorithm and Experiments,

	$p = 1$	$p = 2$	$p = 4$	$p = 8$
B	1,692.25	761.35	378.76	197.43
SS	1,653.39	674.82	330.35	173.54
FS	1,626.74	662.34	319.84	159.25

Table 1: Execution Times for $p = 1, 2, 4, 8$.

Proceedings of Euro-Par'98, Lecture Notes in Computer Science, Springer Verlag.

[3] J.M.Bull, R.W.Ford, T.L.Freeman and D.J.Hancock (1999) A Theoretical Investigation of Feedback Guided Loop Scheduling, Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing, SIAM Press.

[4] C.P.Kruskal and A.Weiss (1985) Allocating Independent Subtasks on Parallel Processors, IEEE Trans. on Software Engineering, vol.11, no.10, pp.1001–1016.

[5] C.D.Polychronopolos and D.J.Kuck (1987) Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, IEEE Trans. on Computers, vol.36, no.12, pp.1425–1439.

[6] D.L.Eager and J.Zahorjan (1992) Adaptive Guided Self-Scheduling, Technical Report 92-01-01, Department of Computer Science and Engineering, University of Washington.

[7] S.F.Hummel, E.Schonberg and L.E.Flynn (1992) Factoring: A Practical and Robust Method for Scheduling Parallel Loops, Communications of the ACM, vol.35, no.8, pp.90–101.

[8] C.W.Reynolds (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, Computer Graphics, 21(4), July 1987, pp. 25-34.

[9] C.W.Reynolds (1999) Steering Behaviors For Autonomous Characters, in the proceedings of Game Developers Conference 1999, San Francisco, California, pp. 763-782.

[10] T.Tabirca, S.Tabirca, L Freeman, T. Yang, Feedback Guided Dynamic Loop Scheduling; A Theoretical Approach, Proceedings of the 3rd Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA 2001), 2001, Valencia, Spain.

[11] T.Tabirca, L.Freeman, S.Tabirca, An $O(\log p)$ Parallel Algorithm for Feedback Guided Dynamic Scheduling, Journal of Parallel Algorithms and Applications, vol 17, no 2, 2002, pp 157-165.

[12] T.Tabirca, L.Freeman, S.Tabirca, A Theoretical Application of the Feedback Guided Dynamic Loop Scheduling, Proceeding of the International Workshop on Clustering Computing, LNCS 2326, Springer-Verlag, 2002, pp. 287-292.

[13] T.Tabirca, S.Tabirca, L Freeman, T. Yang, An Application of Feedback Guided Dynamic Loop Scheduling to the Shortest Path Problem, Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2002), Las Vegas, 2002.

[14] T.H.Tzen and L.M.Ni (1993) Trapezoid Self-Scheduling Scheme for Parallel Computers, IEEE Trans. on Parallel and Distributed Systems, vol.4, no.1, pp.87–98.