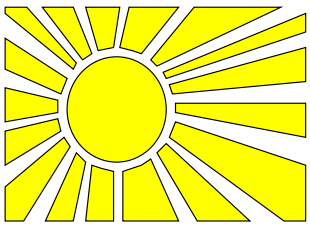
EPCC-SSP-2001-02

Feedback Guided Scheduling for Two-Dimensional Loops

Oleh Olkhovskyy

Abstract

The richest sources of parallelism are loops with independent iterations. These iterations can be distributed to different processors, thus significantly decreasing execution time. However, in such situations, we may meet the imbalance problem , which means that some of the processors finish the work earlier, due to an unequal amount of calculation in different iterations, thus decreasing performance of the algorithm. There are well-known methods aimed at reducing the imbalance in the case of 1-dimensional (1D) loops. The aim of this work is investigation scheduling of 2-dimensional (2D) imbalanced loops on a shared-memory machine. Two possible ways of solving this problem are used here. The first one is based on the reduction of the 2-dimensional loop to a 1-dimensional loop, which allows one to use 1D algorithms, while the second one is based on the use of a 2-dimensional method which directly redistributes 2D loop iterations.



Contents

1 Introduction

The majority of complex mathematical, physical and other problems, can be solved using High Performance Computing. One of the most common examples is given by multiplication of two matrixes. Each element of the outcome matrix is independent of other elements, so their calculation can be redistributed between different processors. Now let us turn to other physical problems which can be parallelised, for instance modelling of the tidal waves in some area. This problem can be parallelised by dividing the area into patches, and giving each processor its own patch. Notice, some parts of the shore are covered with water in high tide and are not covered in low tide, hence these latter parts of the area require less calculations than others, and, respectively, the corresponding processors finish their work earlier than others, which results in imbalance. In fact, many models exhibit this problem. This is why the design of scheduling algorithms, which avoid such an imbalance, is one of the important problems of high performance computing.

2 Background

A lot of scientific and technical models can be reduced to nested loops with independent iterations, which are executed many times, each time with slight changes in numeric values. Thus we have a consecutive outer loop and parallel inner nested loops. In this particular case we consider a 2-dimensional (2D) inner loop. The first possible way is simply to treat it as one-dimensional (1D) loop, which therefore allows us to use standard algorithms for 1D loops. Note, iterations in a loop may be distributed either statically or dynamically. The advantage of a static distribution is that we do not need to expend run-time on the division of loops. But this method does not take into account unpredictable factors, such as an amount of work in every iteration. In a dynamic redistribution time is spent on the division of a loop between processors and on the subsequent communication between these processors. There are well-known 1D imbalance loop scheduling algorithms, which are described in the following sections.

2.1 Guided algorithm

Guided self-scheduling was introduced by Polychronopoulos and Kuck[3]. Guided algorithm provides common patch queue for all processors. When a processor is ready, it receives $\frac{R}{2P}$ iterations, where R is the number of remaining iterations, and P is the number of processors.

2.2 Affinity algorithm

Affinity Scheduling which is used here is Subramaniam and Eager[4] modification of original affinity scheduling, which was suggested by Markatos and LeBlanc[2]. Every processor receives an initial patch, sized $\frac{n}{P}$, where n is size of a parallel loop. Then it removes $\frac{R_i}{P}$ iterations of its local queue and executes them. If processor's queue is empty, it finds a processor with the most amount of work left, removes and executes $\frac{R_{max}}{P}$ part of patch. Every processor keeps track of the amount of iterations it has executed. When all processors finish their work, reinitialisation of initial patch is done, so that each processor will receive the amount of iterations equal to the number of iterations executed during previous time.

2.3 Trapezoid algorithm

Trapezoid self-scheduling (TSS) was introduced by Tzen and Ni[5]. Again the algorithm provides common patches queue, as in the guided algorithm. Each free processor receives and executes a patch from this queue . This time the size of a patch is decreased linearly, from f to l. The number of patches is $N = \frac{2n}{f+l}$. Every time we get a new patch, the patch size is decreased by $\delta = \frac{f-l}{N-1}$. In this particular implementation we use $f = \frac{n}{2P}$ and $l = 1$.

3 Feedback Guided Loop Scheduling

This 2-dimensional algorithm was proposed by Bull[1]. The loops iterations are divided into P patches, and are distributed between processors. Each processor executes its iterations, keeping a track on time of executing of the whole patch. Then dividing this time by the number of iterations in patches we get the mean load per iteration. Next, we find new boundaries, based on equipartitioning of the area under the mean load per iteration. New boundaries are formed as follows: first we divide the area in the x-direction, so that aggregate time in the left and right parts is in the ratio $\left\lceil \frac{P}{2} \right\rceil$. Then with each of new patches we repeat the procedure, cutting each patch in the direction perpendicular to the previous one, until we have P patches. A second way is similar, except that we choose cut direction in such a way that new patches are as square as possible.
For benchmarking we also use a 1-dimensional version of FGLS.

4 Guided 2D

This algorithm is based on the guided algorithm, described above, with some changes. The size of a patch is $\frac{R}{2P}$. Dividing the area into rectangles of exponentially decreasing size is quite a difficult task. Here we use an algorithm, that allows us to get almost square patches, with almost exponentially decreasing size.
The first patch $\left(\text{sized } \frac{R}{2P}\right)$ is square. The next few patches are of the same width, but decreasing heights, and are under the first one. The remained of the height, is either distributed between these patches, or forms a new patch, if it is close to the patch size. This operation is repeated for the rest of the area but in alternating direction.
Figure 1 shows an example of dividing an area of the size 200x200 for four processors. Darker colour show patches build in x-direction, lighter colour shows patches build in y-direction.

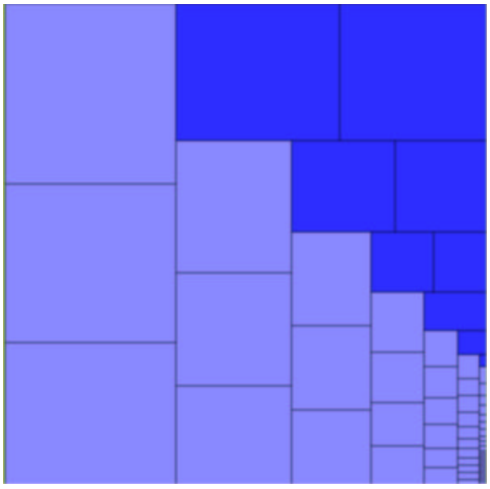


Figure 1 : Guided 2D

5 Load

The experimental workload consists of an imbalance load and communication. The load distribution can be iteration-dependent, so we can test the efficiency of the the history based algorithms. Each algorithm calls a routine, passing to it an area and iteration. For each point from the area, this routine executes communication code, then finds the imbalance time associated with this area and is paused for time interval $\frac{time}{const}$. Time associated with the area is the sum of times associated with each point. The communication routine is described below.

5.1 Communication

Each point of an area is initialised with some value.
.On every iteration, each point is the mean value of its 4 nearest neighbours.

5.2 Gaussian load

Each point has associated time,

$$time = \exp^{-\frac{(x-pos_x)^2+(y-pos_y)^2}{width^2}}$$

where $pos_x = center_x + radius * \sin(\pi * iteration/period)$,
$pos_y = center_y + radius * \cos(\pi * iteration/period)$.
The load moves around the center counter-clockwise, starting from the bottom-center position.

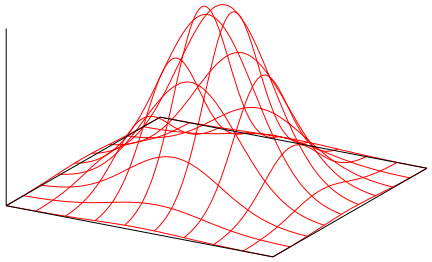


Figure 2 : Gaussian load

5.3 Pond load

$time = 1$ if the point is inside a circle of radius $R + A * \sin\left(\frac{2*\pi*iteration}{period}\right)$ and 0 otherwise. As it is clearly seen from the equation, the radius changes sinusoidally with iteration.

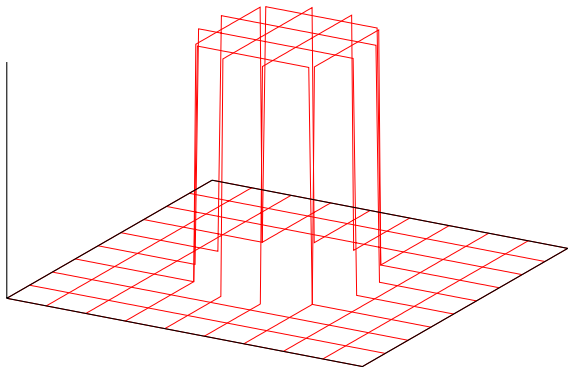


Figure 3 : Pond load

5.4 Ridge load

$$time = \exp^{-\frac{\left(x*\cos\theta + y*\sin\theta - \left\{\frac{iteration}{period}\right\}*(xpoints*\cos\theta + ypoints*\sin\theta)\right)^2}{width^2}}$$

This imbalance simulates a solitary. wave, which moves from the bottom left conner with each iteration.

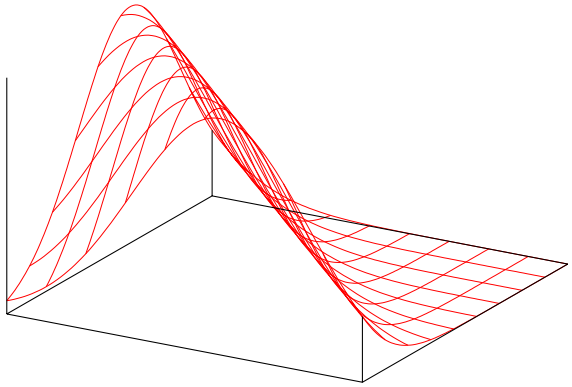


Figure 4 : Ridge load

5.5 Sinusoidal standing wave load

$$time = \sin^2\left(\frac{3*\pi*x}{xpoints} + offset\right) * \sin^2\left(\frac{3*\pi*y}{ypoints} + offset\right)$$

Where $offset = 2 * \pi * \frac{iteration}{period}$

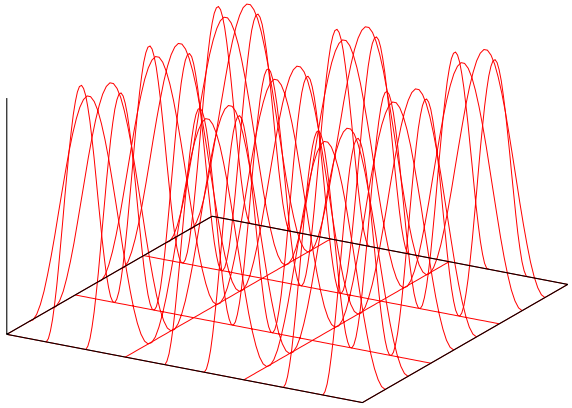


Figure 5 : Sinusoidal load

6 Results

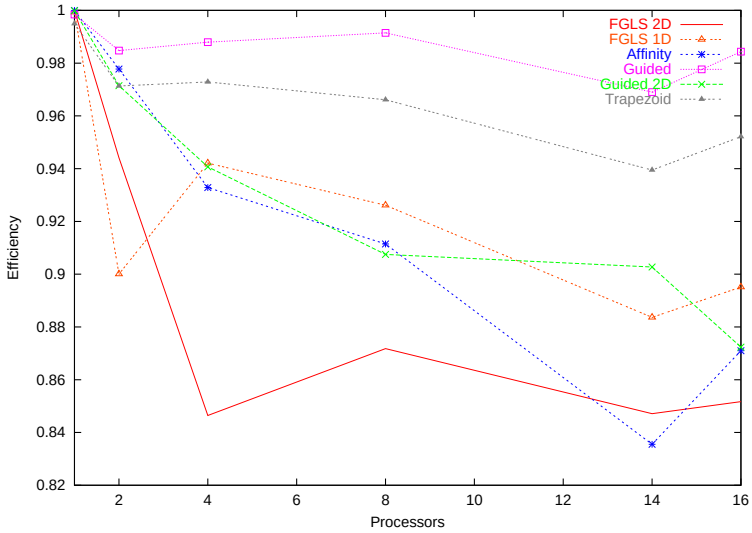


Figure 6 : Gauss load

Figure 6 shows the results of running the benchmarks with Gaussian load, with clear imbalance. The difference between algorithms is not large, the efficiency is in the range 0.85-0.98.
A 1D algorithm distributes iterations by lines, and hence in each bunch of iterations there are points with high load and low load. Thus the corresponding imbalance is spread more than in the case of 2D algorithm.

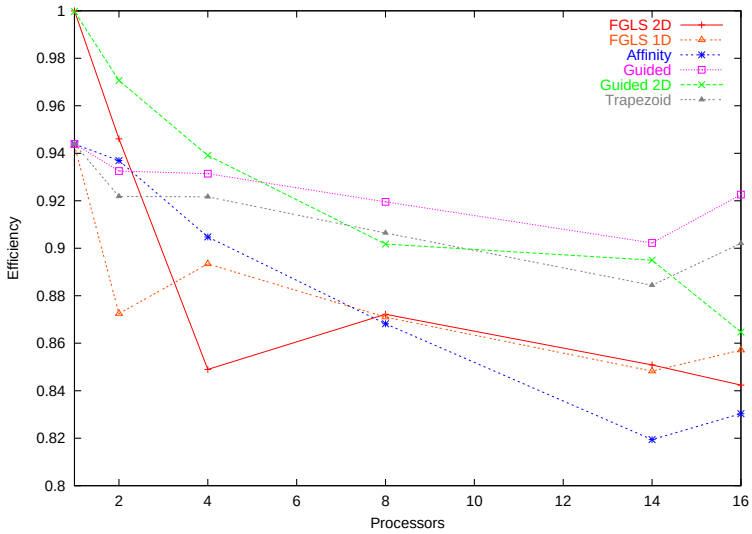


Figure 7 : Gauss load

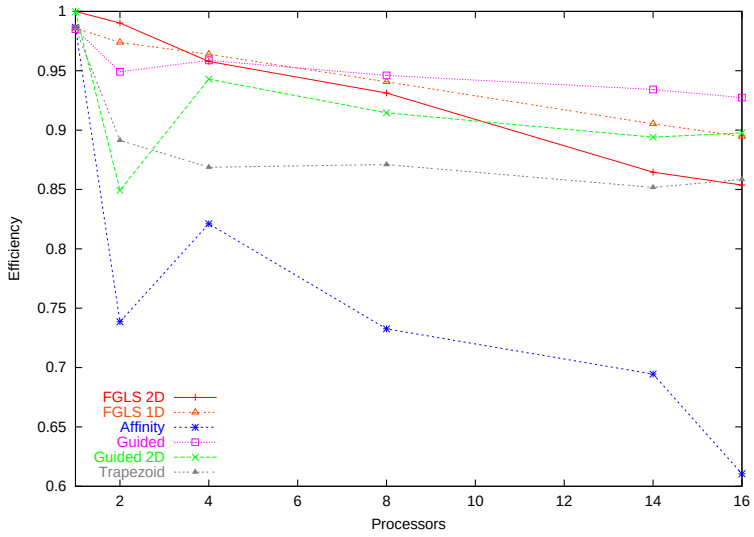


Figure 8 : Pond load

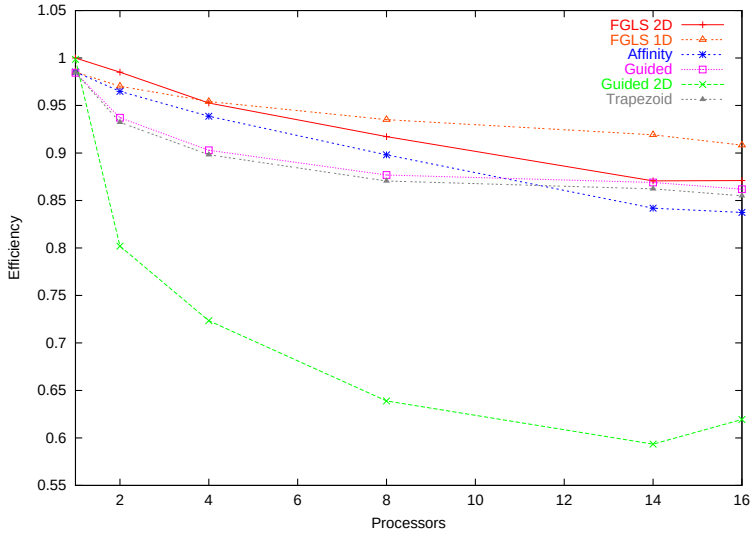


Figure 9 : Ridge load

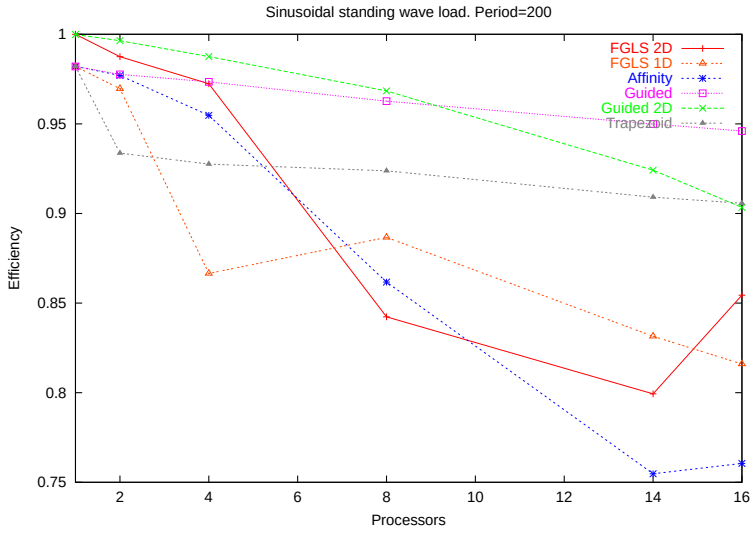


Figure 10 : Sine load

Figures 7-10 shows the results of running the benchmarks with different loads, with imbalance$\gg$communication. Again the difference between algorithms is not large.
Because communication is still not considerable, figure 7 does not differ a lot from figure 8, also this time efficiency range is smaller: 0.83-0.93

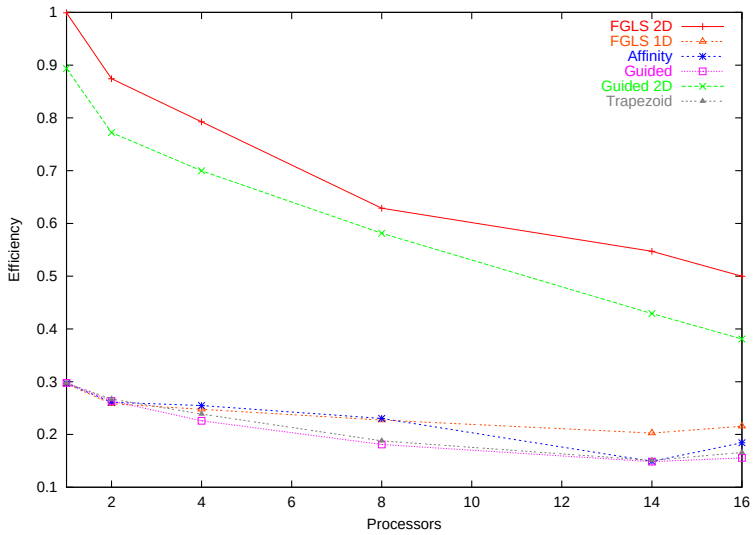


Figure 11 : Gauss load

Figure 11 shows the results with approximately the same time of imbalance and communication. Here the advantage of 2D algorithms is clearly seen. A 2D algorithm creates patches which are more square than 1D does, and hence needs less data from outside its boundaries. This means that most data which is used by the processor can be cached. From 1D algorithms FGLS is ahead, because it produces only $\boldsymbol{P}$ patches, hence it has fewer loops which means that it has lower loop overhead. This is also the advantage of FGLS 2D over Guided 2D.

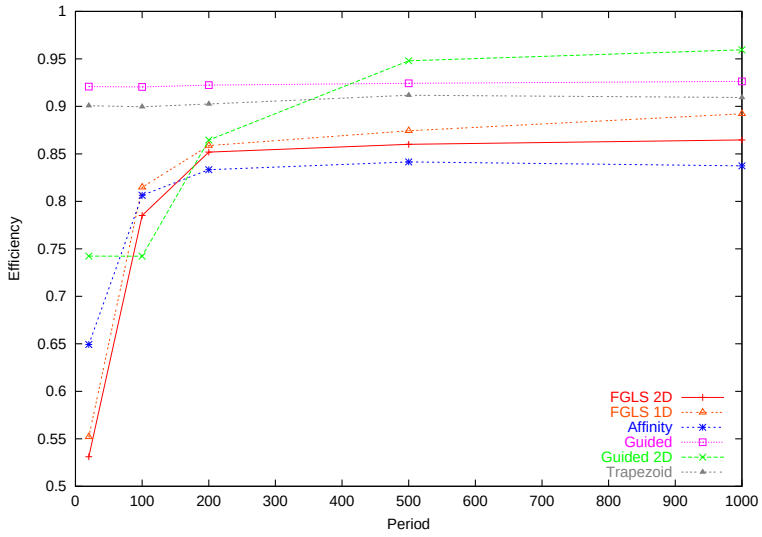


Figure 12 : Varying period

Figure 12 shows algorithms behaviour with dynamic imbalance. The higher period is the slower imbalance changes with iterations. Algorithms which are not based on history, which means that they do not depend on previous executions, shows they do not depend on a period. However algorithms with history are less efficient when imbalance changes more rapidly. The exception here is Guided 2D algorithm, which shows quite strange behaviour. However this is explained by the fact, that when the period is more than 100 all imbalance is located in the lower right conner, where the guided 2D has the smallest patches.

7 Conclusion

In the imbalance only case, 1D algorithms are more efficient, since they distribute iterations by lines, not by rectangles, as 2D algorithms do. Hence each bunch of iterations has points with high load and low load. The corresponding imbalance is spread more than in the case of 2D algorithm.
The FGLS 1D algorithm is not the most efficient algorithm, but still shows quite good results, especially with ridge load.
In situation when communication time is considerable in comparison with imbalance time, 2D algorithms are ahead of 1D algorithms, due to the fact that they use patches close to squares. Since FGLS produces only $\boldsymbol{P}$ patches it is more efficient then Guided 2D. For high efficiency, both FGLS 1D and 2D require slowly changing imbalance; this can be clearly seen from Figure 12.

References

[1] Bull J.M. Feedback Guided Dynamic Loop Scheduling: Algorythms and Experiments. in Proceedings of EuroPar '98, Lecture Notes in Computer Science vol.1470,Berlin, 1998.

[2] Markatos E.P., LeBlanc T.J. Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors. IEEE Transactions on Paralel and Distributed Systems Vol.5, No.4, April 1994.

[3] Polychronopoulos C.D., Kuck D.J. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Transactions on Computers, Vol. 36,No 12, December 1987.

[4] Subramaniam S., Eager D.L. Affinity Scheduling of Unbalanced Workloads. University of Saskatchewan Saskatoon.

[5] Tzen T.H., Ni L.M. Dynamic loop scheduling dor shared-memory multiprocessors. In Proc. 1991 International Conference on Parallel Processing, August 1991.

My name is Oleg. I have finished 2nd year in Cybernetics faculty at University of Kiev. This is in Ukraine, in Europe.

Supervisor:
Mark Bull.