5DV152/VT15: Lab 4

Christer Jakobsson (870310-8533)

March 9, 2015

1 Experiment design

The experiments where designed in the way that the file *mandel-basic.c* where modified so that there where no image produced. Since the image is not used, the experiments will be done faster if the image isn't made on every repetition. Modification where made in the file *common.c*, to be able to get experiment data.

For one set of experiments the code were modified so that there where two extra for loops, one to set the environment variable to how many cores should be used. This iterated from 1 to 48 doing runs with different number of cores used. The other (inner) for loop was used to do repetitions on the calculations 10 times to get more experiment data.

So the first experiment where done so that i did ten runs with each number of processors (1 to 48), since there is an implicit barrier at the end of the parallel loops, then printed the time it took for each run and calculated the means and standard deviations with the data i gathered using *octave*.

In the second experiment to analyze the load-imbalance, a similar experiment was done, with the difference that *nowait* were used so that the cores did not synchronize after the parallel loops. Each core printed their own execution time. For each repetition i summarized each threads difference from the thread with max time minus its own execution time. Then created a list with a sum of the total idle time for each repetition.
Then i calculated the mean of all the total idle times and plotted a graph with *errorbars*.
The experiment where run with cores 1-48 and 50 repetitions per core.

For each repetition.
sum(maxExectutionTimeInRepetition - coresExecutionTime

Paremeters for mandel-basic

- *Height:* 700
- *Width:* 600
- *Pixelsize:* 0.00001
- *Centerreal:* 0.32
- *Centerimage:* 0.039
- *Maxiterations:* 800

2 Results

Shows the results from the experiments by showing diagrams with calculations from the experiments.

2.1 Mean execution times

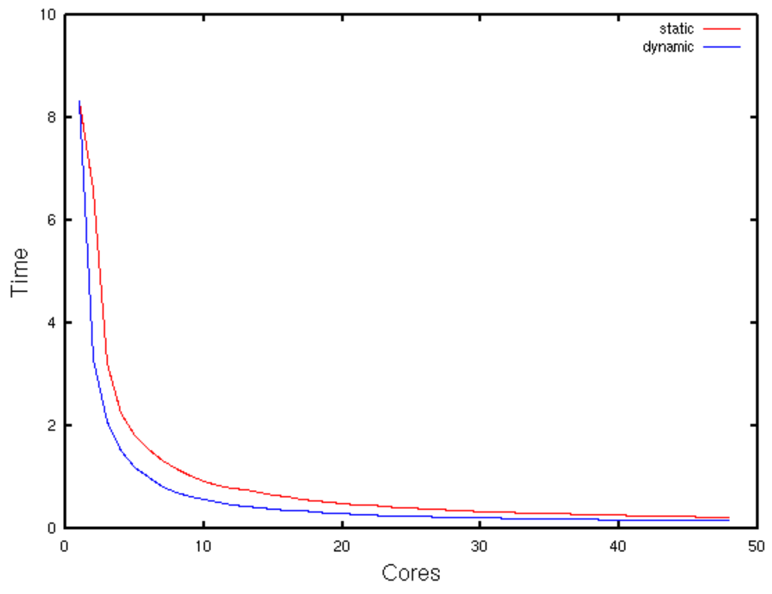

Figure 1: *Mean execution time: Static and Dynamic scheduling*

In Figure 1 we see that dynamic seems to have a lower execution with number of cores up to 48.

2.2 Mean speed-up

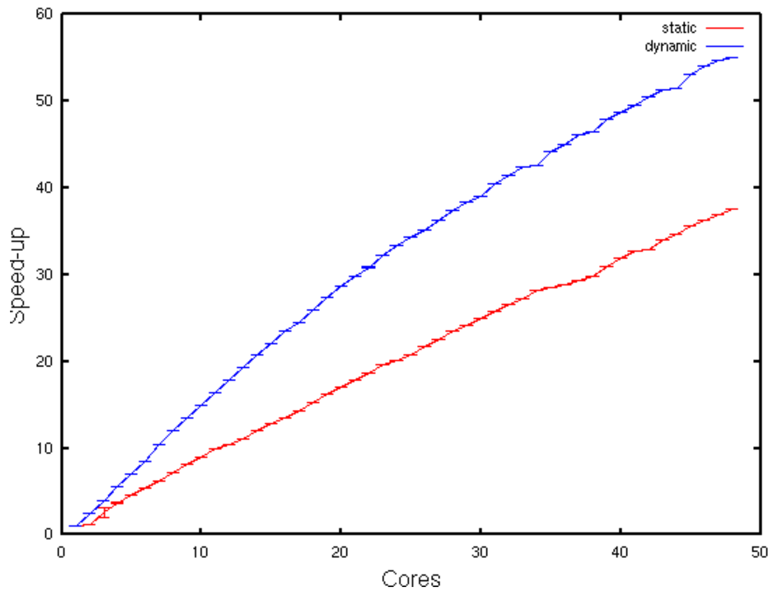

Figure 2: *Mean speed up: Static and Dynamic scheduling*

Shows that the curve for both static and dynamic follows the same pattern, but that dynamic have a higher speed-up.

2.3 Mean load imbalance

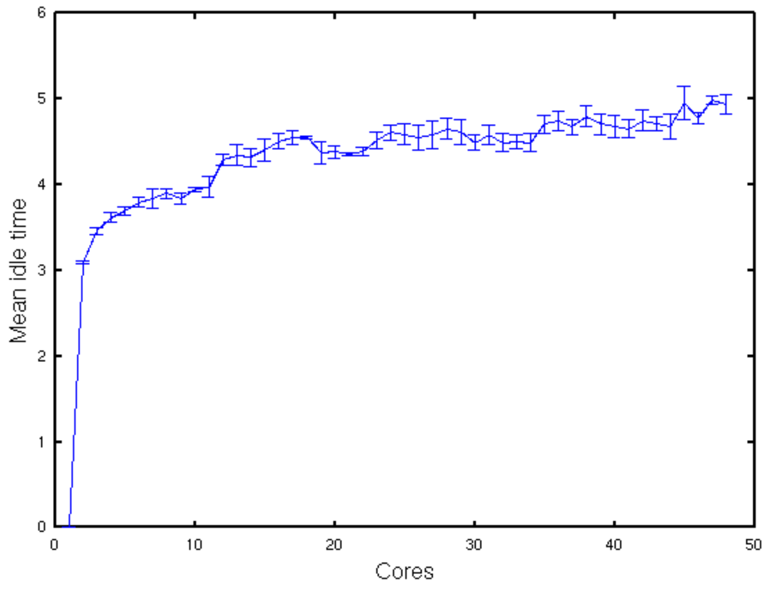

Figure 3: *Mean idle time, static scheduling*

3 Conclusion

3.1 Scalability static vs dynamic

Seen from the graphs in 2.1 and 2.2 there seems to me that dynamic is in general faster then static scheduling to execute this particular image with the settings i have chosen. In the aspect of scalability there seems that the speed-up does not converge up-to 48 cores, but the execution time seems to converge with the x-axis at 30 cores.

Both static and dynamic scheduling seems to scale good when using many cores, dynamic scheduling scales better seen in 2 as its speed-up is more then 50 times with 48 cores, static scheduling speed-up is at around 35 using 48 cores. Although that dynamic seems to scale better, static does also scale impressively well.

Static vs dynamic

	Static	Dynamic
Lowest execution times	No	Yes
Best speed-up	No	Yes
Converging with x axis?	No	No

Both static and dynamic scheduling does increase in effectiveness when using up to 48 cores, Although static scheduling seems to lead to higher execution times, from Figure 1 there seems like static scheduling is still reaping benefits from using more and more cores, while dynamics execution speed seem to have converged at around 30-40 cores. This might be because static scheduling is optimized at compile time and therefore utilizes the cores in a more efficient manner when there is many. While dynamic sheduling might lead to poor use of cores since its a out of order, first come first served method.
Dynamic scheduling seems to scale better then static scheduling.

3.2 Load imbalance

The load imbalance increases as the number of cores used increases, looking at the graph in Figure 3 the total idle time increases much even on a low number of cores. There seems like the load imbalance always increases when increasing used cores, which seems likely. Looking at the standard deviations there seems like the idle times for each repetition fluctuates quite much, which means that the work get distributed in a random fashion. Giving different total idle times for each repetition.
My initial thought was that there would be some load-imbalance, but not that there would be this much time wasted as there is. Interesting to see that something is not as optimal as one thinks.