Observations/Results

Experimented with few Openmp programs, varying the number of threads, and the scheduling pattern.

Varying the number of threads

Serial PI Program

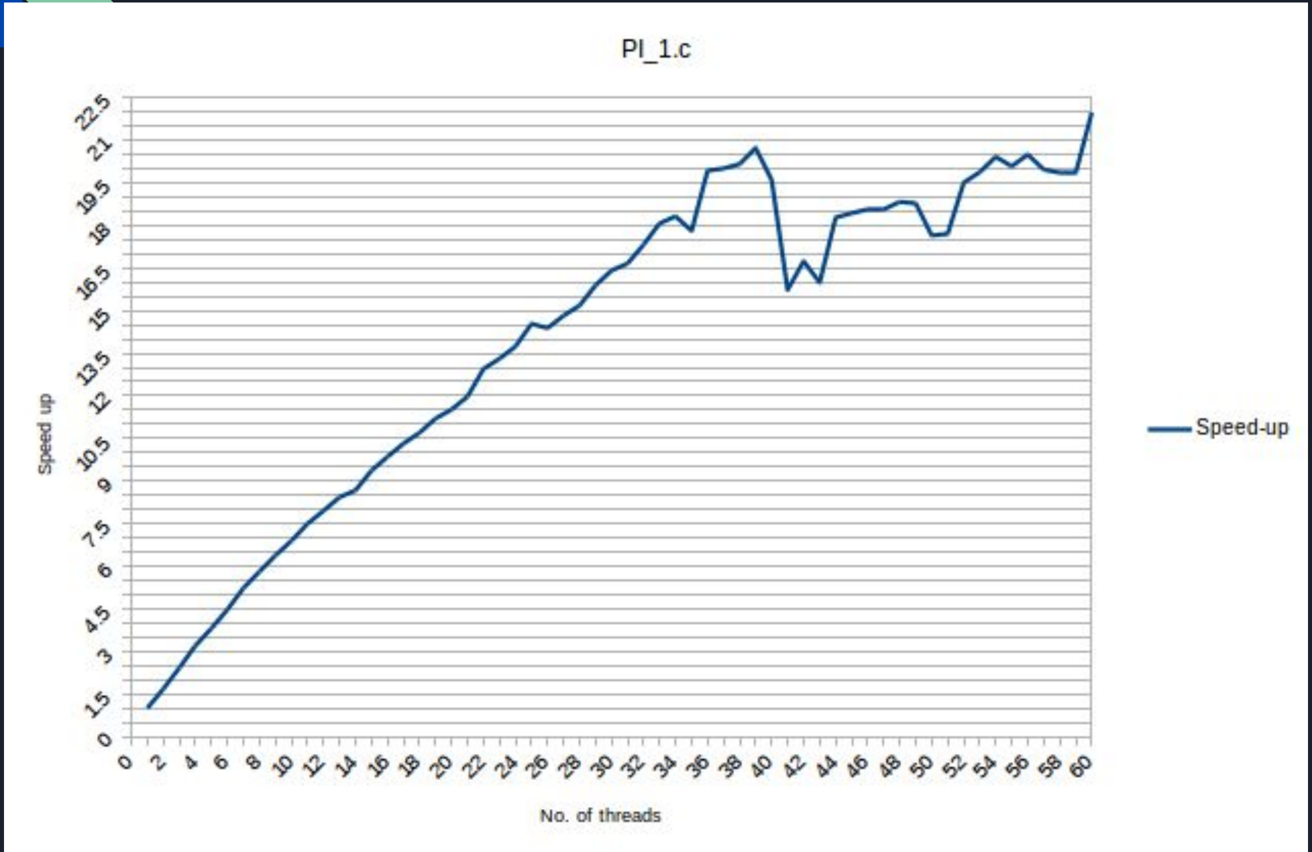
```
static long num_steps = 100000;
double step;
int main ()
{
int i; double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
for (i=0;i< num_steps; i++){
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

Tried out 3 ways to parallelize - Program 1

```
#pragma omp parallel
        {
                int i;
                double sum;
                double x;
                int id = omp_get_thread_num();
                int num_threads =  omp_get_num_threads();
                if(id == 0)
                        nthreads = num_threads;
                for (i=id, sum = 0.0; i< num_steps; i+=num_threads){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
                }
                sum = sum * step;
#pragma omp atomic
                pi += sum;
        }
```

'Atomic' section provides mutual exclusion for the threads.

Performance



Optimum :
39 threads

Program 2

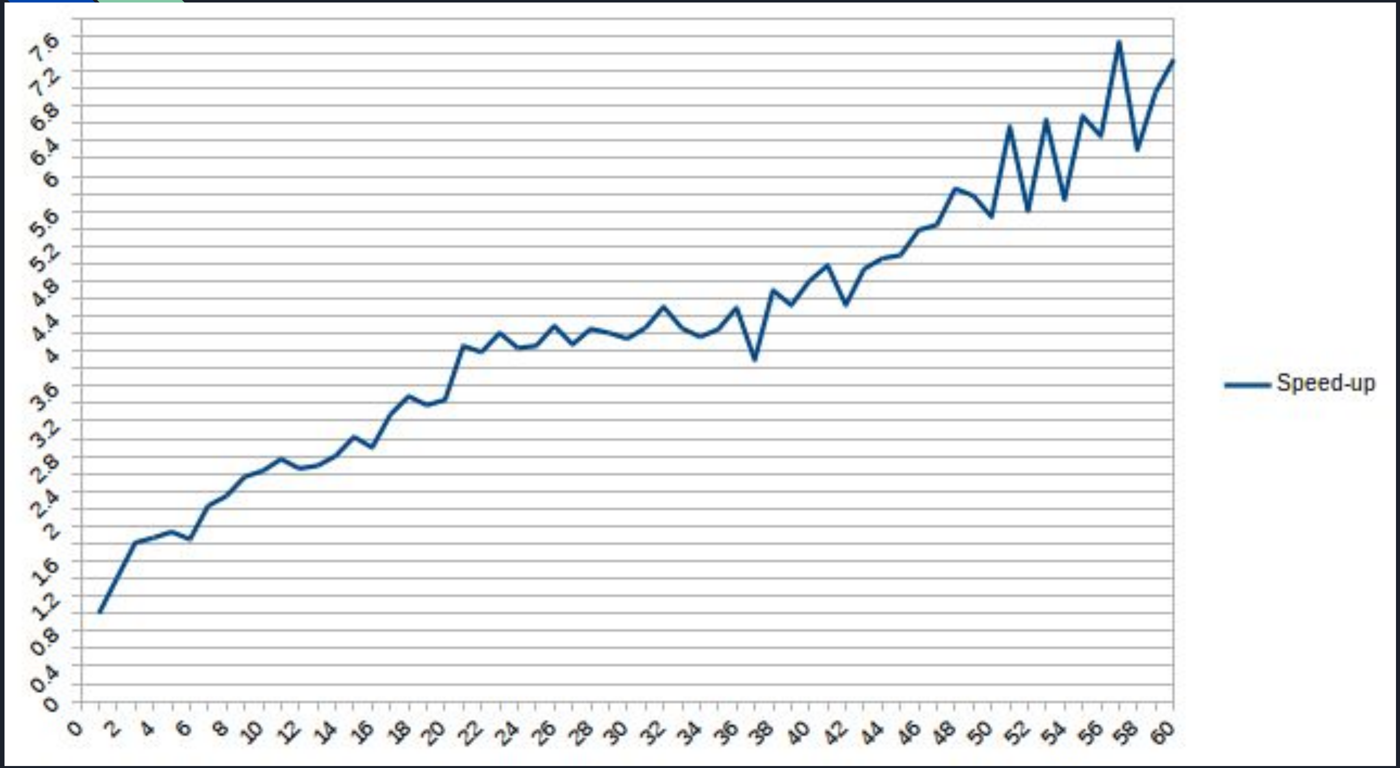
```c
#pragma omp parallel
	{
		int i;
		int id = omp_get_thread_num();
		int numthreads = omp_get_num_threads();
	//	printf(" num_threads = %d",numthreads);
		double x;

		sum[id] = 0.0;

		if (id == 0) {
			nthreads = numthreads;
			}

		for (i=id;i< num_steps; i+=numthreads){
			x = (i+0.5)*step;
			sum[id] = sum[id] + 4.0/(1.0+x*x);
		}
			//printf("Fdf");
	}

	for(full_sum = 0.0, i=0;i<nthreads;i++)
		full_sum += sum[i];

	pi = step * full_sum;
```

Here, an array is used for the variable 'sum'.

Array elements are continuous in memory, so they will share the same cache line.

In this program, there is bad cache-hit ratio, performance is not as good as before.

Performance



Optimum :
57 threads

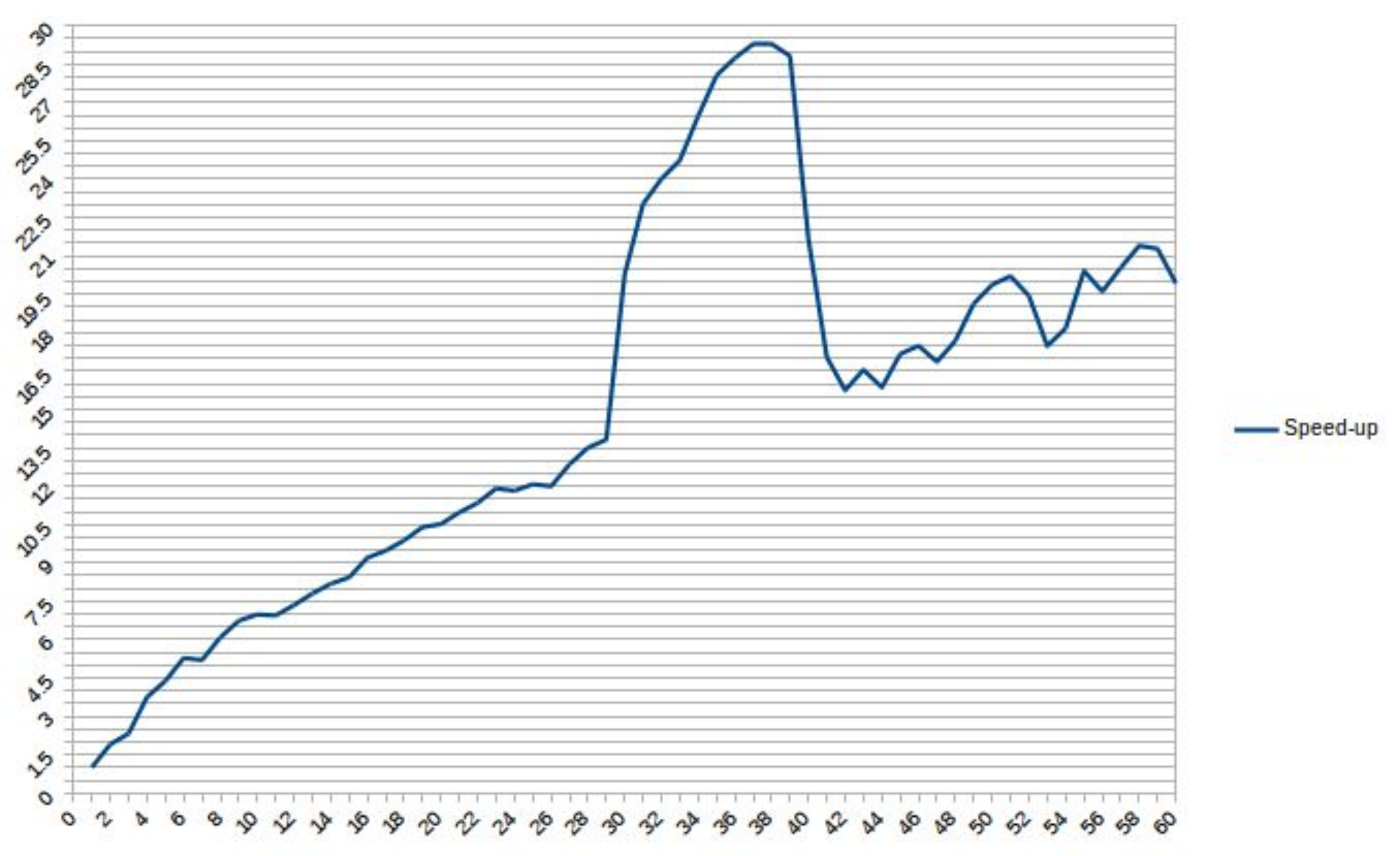
Program-3

```
#pragma omp parallel
        {
                int i;
                double x;
                int id = omp_get_thread_num();
                int num_threads =  omp_get_num_threads();
                if(id == 0)
                        nthreads = num_threads;
                for (i=id, sum[id][0] = 0.0; i< num_steps; i+=num_threads){
                        x = (i+0.5)*step;
                        // This stores the partial sum from each thread
                        sum[id][0] = sum[id][0] + 4.0/(1.0+x*x);
                }
        }

        int i;
        double pi =0.0;
        for(i = 0; i < nthreads;i++)
                pi += sum[i][0] * step;
```

Here, we padded the sum array. Now it is a double array so each sum value is in a different cache line.

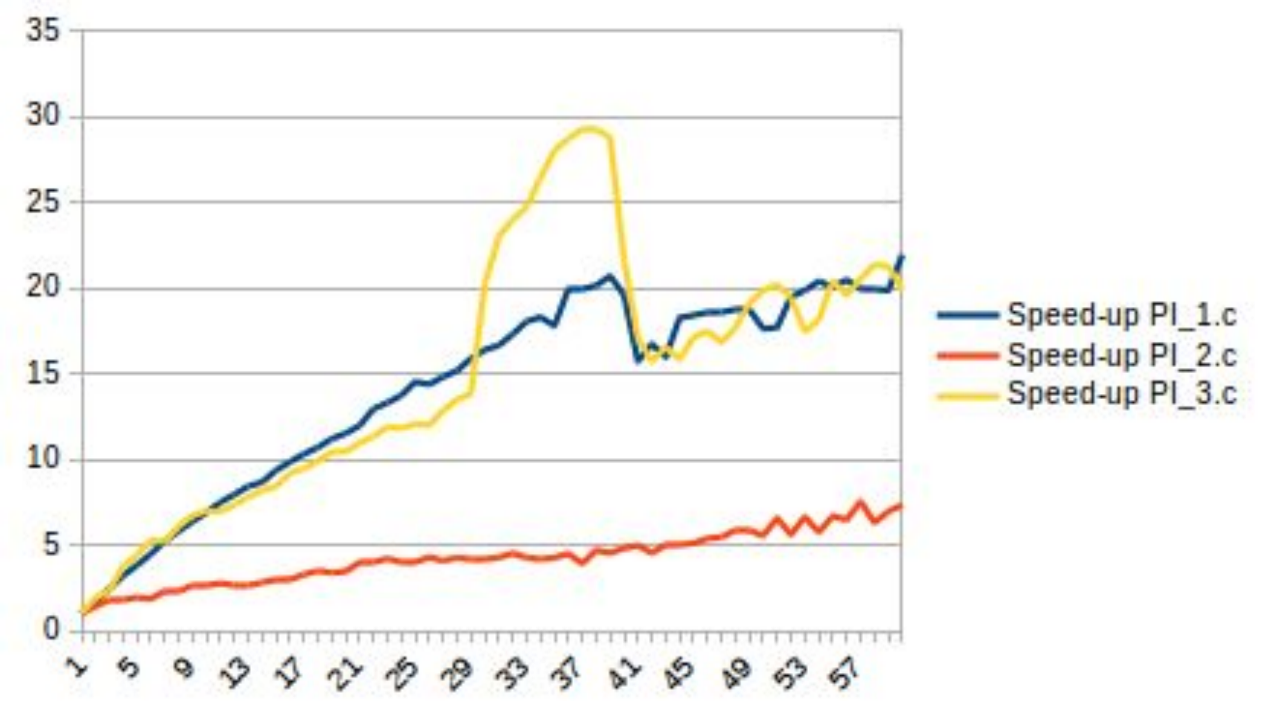
Performance much better than before version, which had a bad cache-hit ratio.

Performance



Optimum : 38 threads

Comparison all three



PI_1.c -> Using atomic section

PI_2.c -> Using array (bad cache-hit ratio)

PI_3.c -> Using padded array

Varying the scheduling strategy

PI_Loop_1.c

```c
int main (int argc , char ** argv)
{
		int i;
		unsigned int num_threads = atoi(argv[1]);
		double x, pi, sum = 0.0;
		double start_time, run_time;

		step = 1.0/(double) num_steps;
		sum = 0.0;
		omp_set_num_threads(num_threads);
		start_time = omp_get_wtime();
#pragma omp parallel
{
//#pragma omp single
//		printf(" num_threads = %d",omp_get_num_threads());

#pragma omp for reduction(+:sum) schedule(static,CHUNK)
		for (i=1;i<= num_steps; i++){
			x = (i-0.5)*step;
			sum = sum + 4.0/(1.0+x*x);
		}
}
		pi = step * sum;
```

PI_Loop_2.c

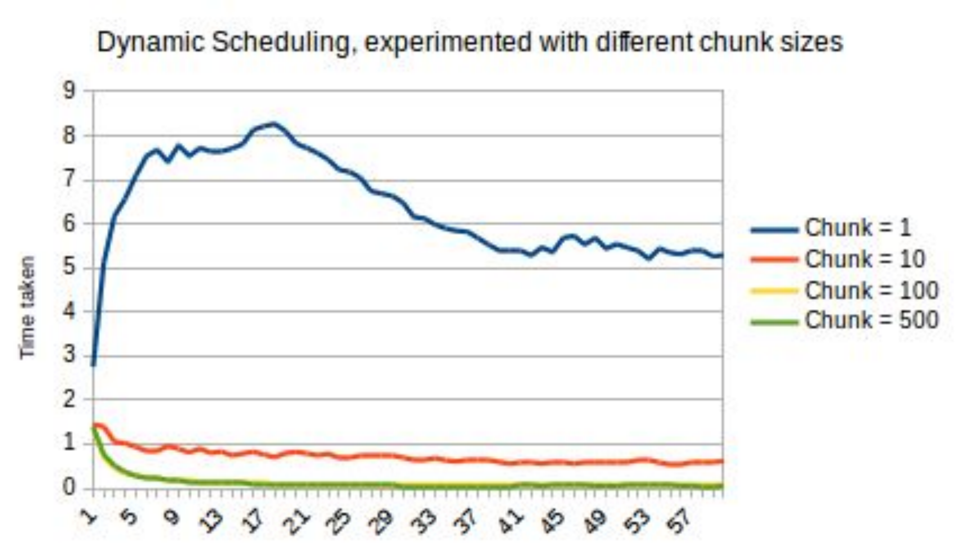
```c
#pragma omp parallel
{
		double x;
//#pragma omp single
//		printf(" num_threads = %d",omp_get_num_threads());

#pragma omp for reduction(+:sum) schedule(static,CHUNK)
		for (i=1;i<= num_steps; i++){
			x = (i-0.5)*step;
			sum = sum + 4.0/(1.0+x*x);
		}
}
		pi = step * sum;
```

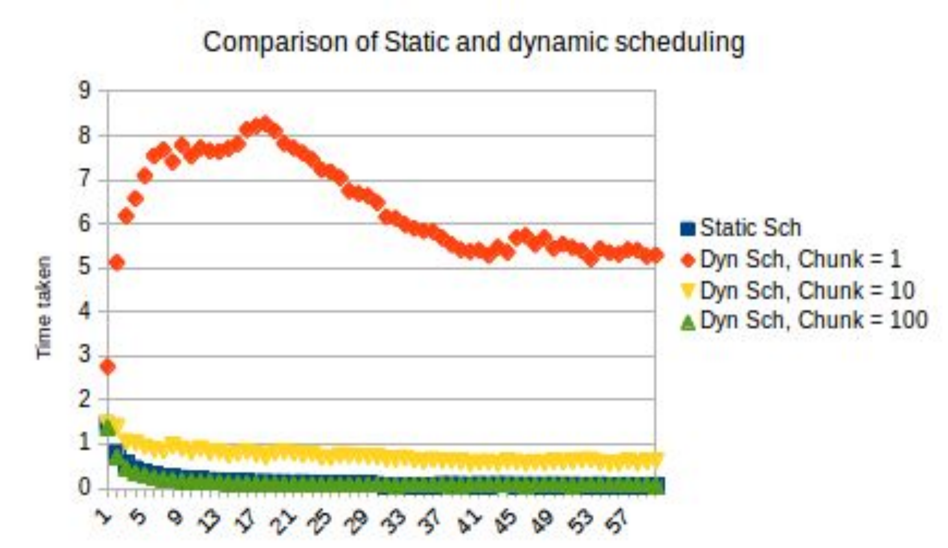
In the 2nd program, each thread given its own local copy of variable 'x') - this speeds up the program by about 10 times

Different types of scheduling we tried

- Default - ***STATIC***. (each thread is assigned a set of loop iterations in round-robin fashion)
- Not always optimal (for e.g. when different iterations take different amounts of time)
- In dynamic, one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasn't been executed yet.
- However, there is some overhead to dynamic scheduling. (After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.)
- We can split the difference between static and dynamic scheduling by using *chunks* in a dynamic schedule.

PI_Loop_2.c

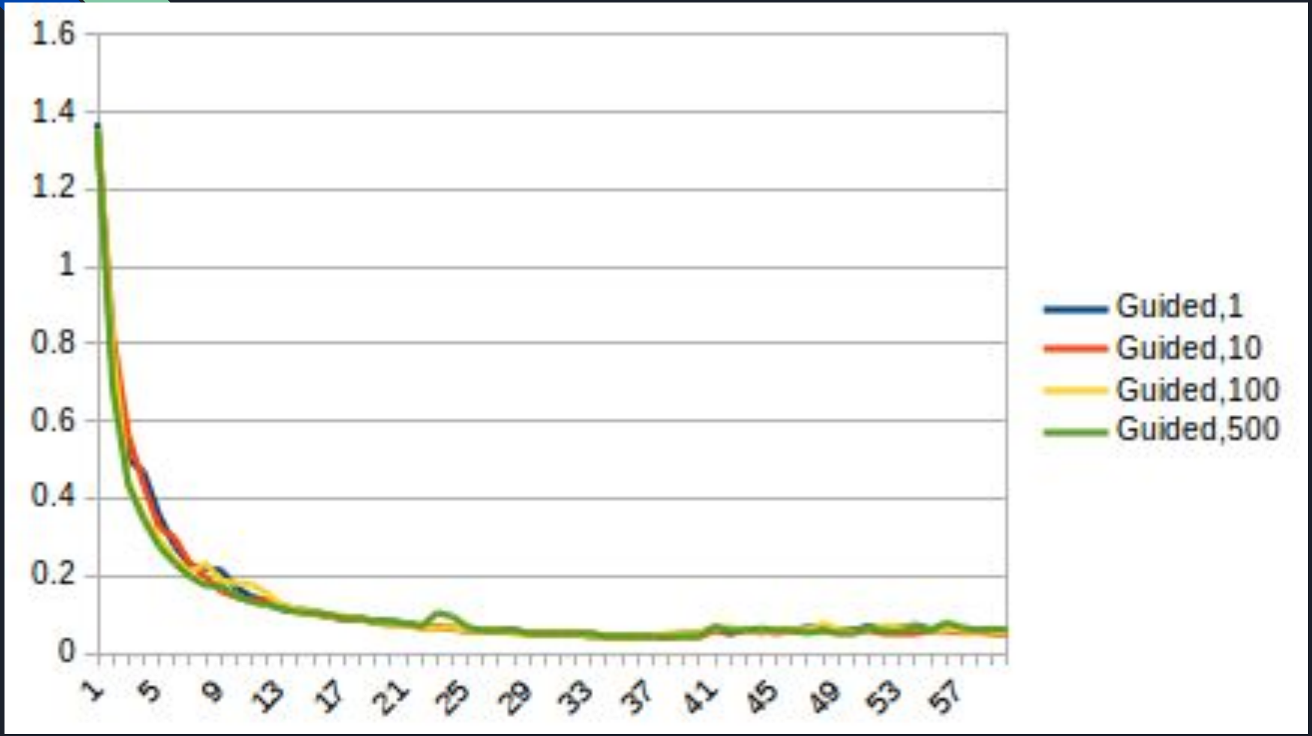

Dynamic Scheduling, experimented with different chunk sizes
Time taken
0
1
2
3
4
5
6
7
8
9
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
Chunk = 1
Chunk = 10
Chunk = 100
Chunk = 500


Comparison of Static and dynamic scheduling
Time taken
0
1
2
3
4
5
6
7
8
9
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
Static Sch
Dyn Sch, Chunk = 1
Dyn Sch, Chunk = 10
Dyn Sch, Chunk = 100

Analysis of static and dynamic Scheduling

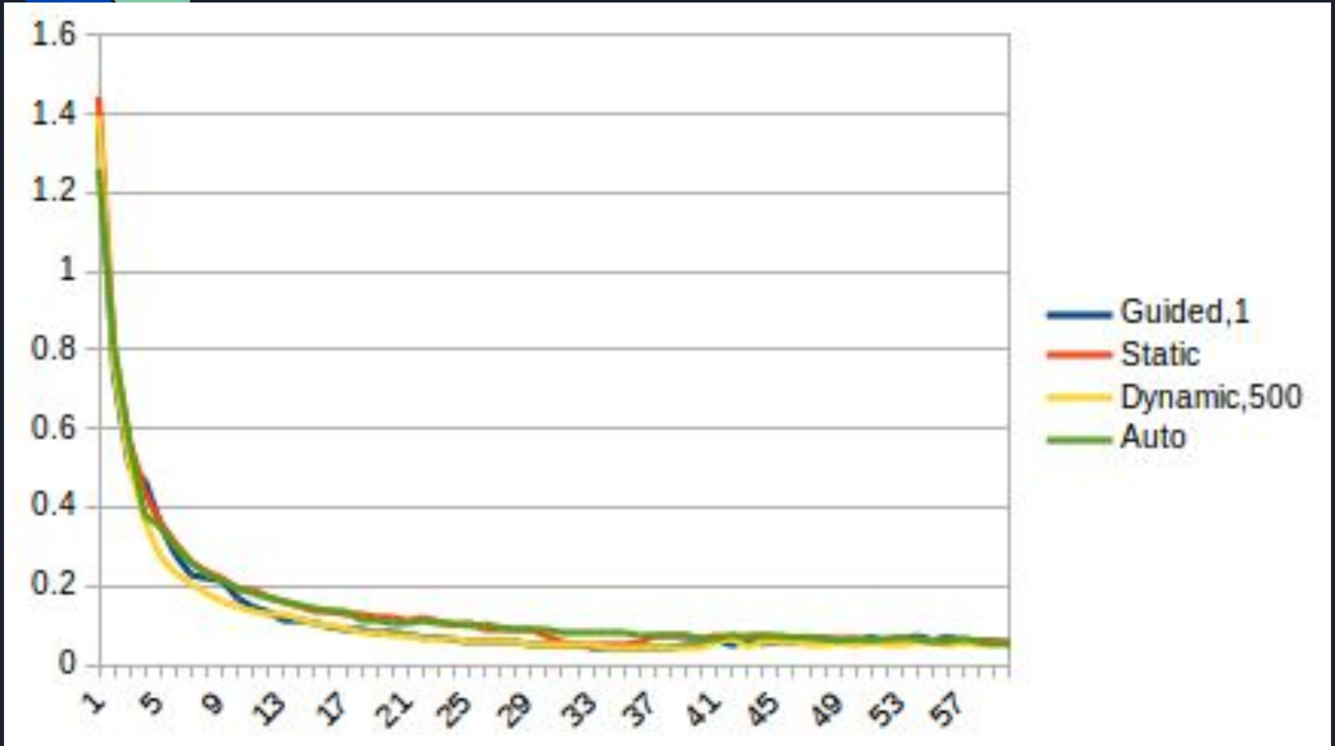
- Increasing the chunk size makes the scheduling more static.
- Decreasing the chunk size makes it more dynamic in nature.

PI_Loop_2.c -> guided scheduling



Very little change observed as we changed the chunk size for guided scheduling.

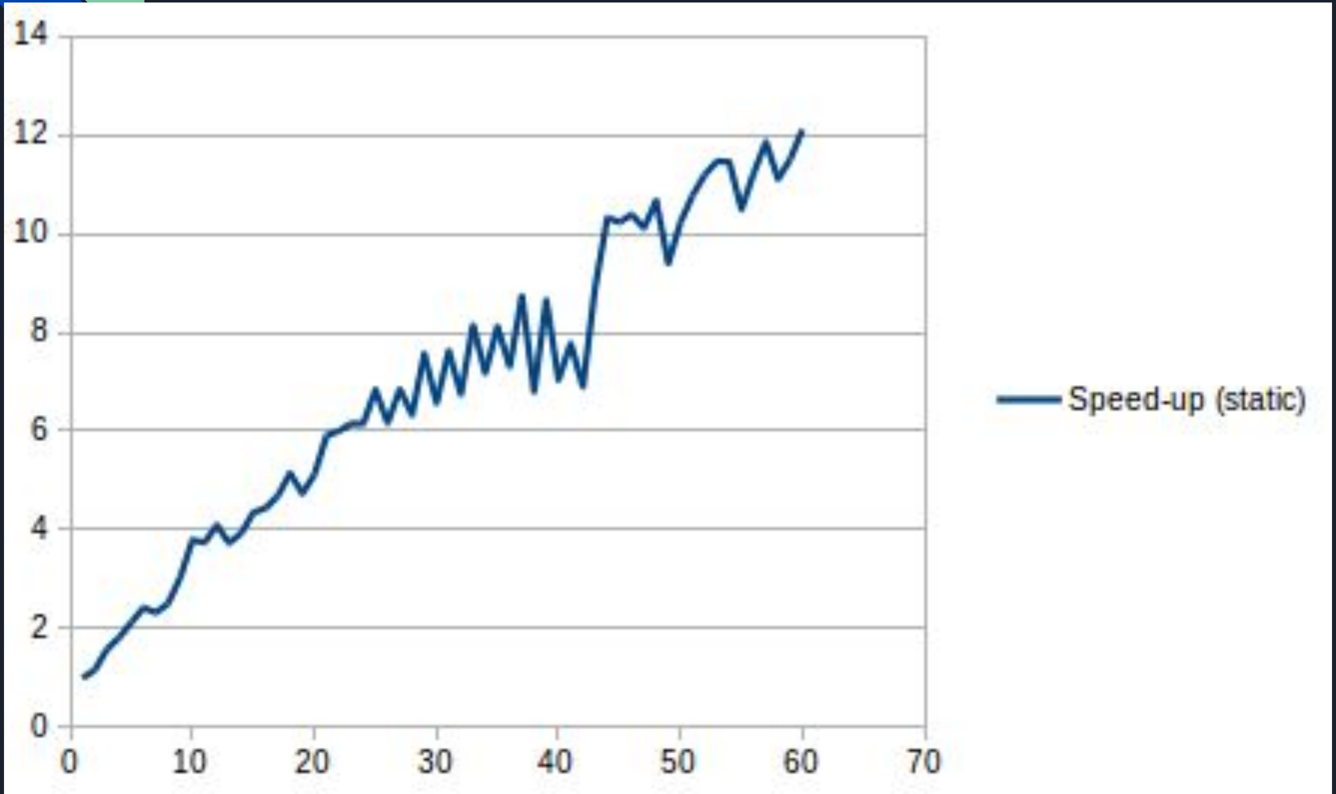
PI_Loop_2.c -> which is best?



In this program, all iterations take the same time so dynamic scheduling is not useful and it adds unnecessary overhead also.

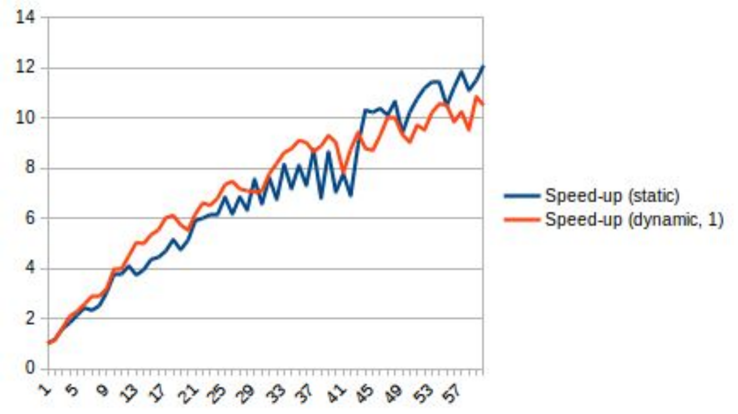
Auto/static scheduling give the optimal time.

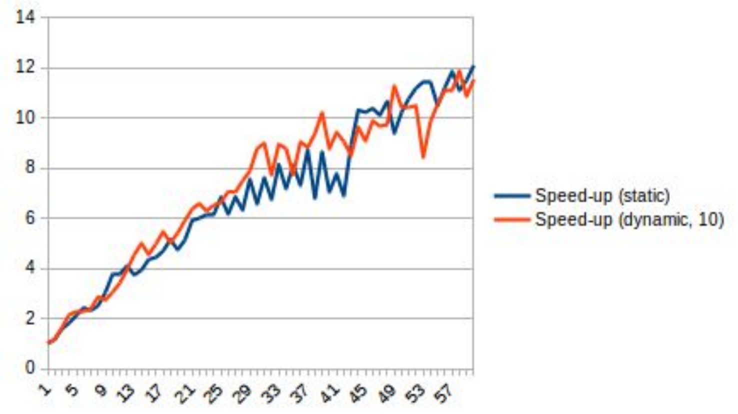
Mandel.c

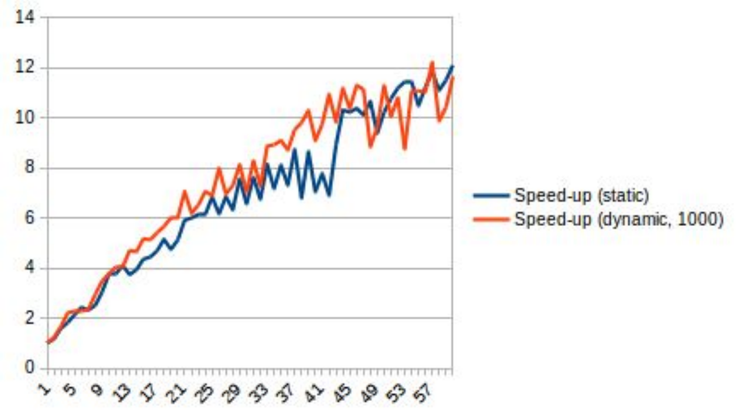


Optimal : 60 threads

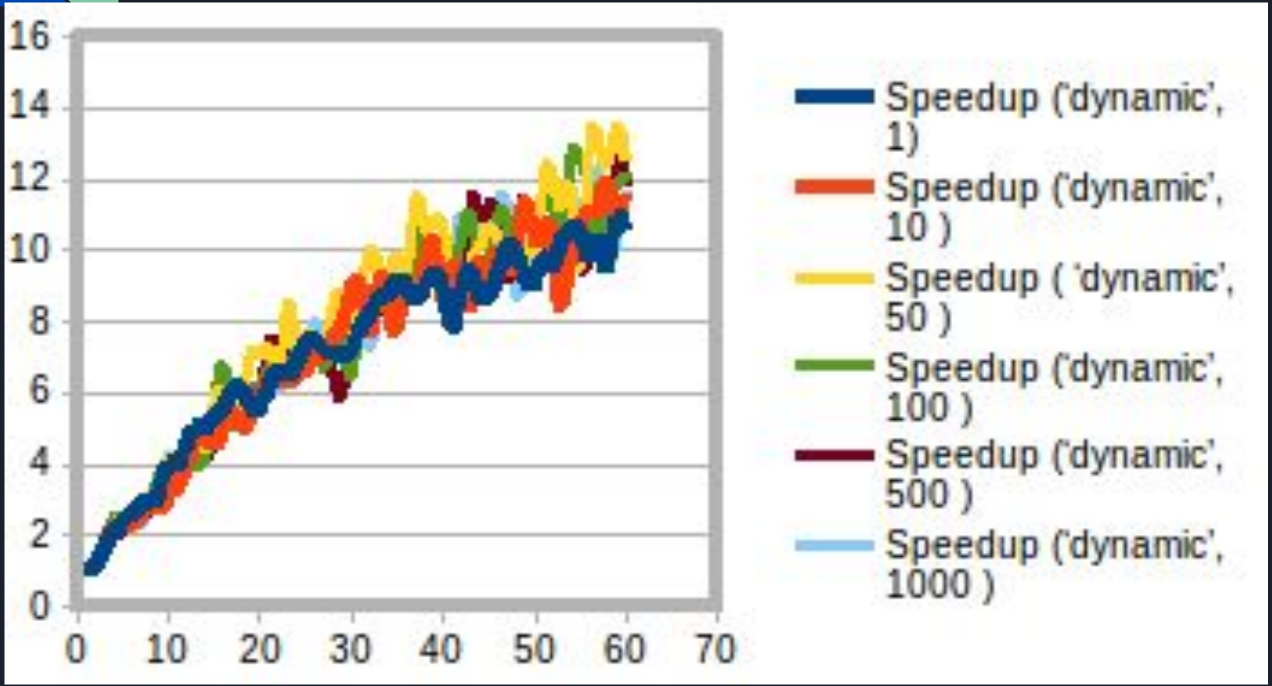
Mandel.c


14
12
10
8
6
4
2
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57
Speed-up (static)
Speed-up (dynamic, 1)


14
12
10
8
6
4
2
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57
Speed-up (static)
Speed-up (dynamic, 10)


14
12
10
8
6
4
2
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57
Speed-up (static)
Speed-up (dynamic, 1000)

Mandel.c



We observed that, in this program, up till a particular chunk size (50) the speed up, in general, increased with increase in chunk size. After that the speed up remained almost same.

Mandel.c (guided scheduling)