Shared Memory Programming with OpenMP

Roshan Pasupathy, 26593165, rp3g13@soton.ac.uk

Submitted as part of Advanced Computational Methods 2 – FEEG6003

Table of Contents

Introduction

Over the years, scheduling strategies for parallel programming have been extensively researched and benchmarked [1-3]. The aim of these scheduling strategies is to optimise workload distribution between threads – load balancing. However, there is "no one size fits all" solution for scheduling as the choice of strategy greatly depends on the nature of iterations to be parallelised as well as on the threads they are performed on. In this report a number of OpenMP scheduling strategies have been compared for parallelising two functions with unique loop characteristics. An Affinity scheduling strategy is also developed and compared to the best scheduling clauses.

OpenMP Scheduling Clauses

As stated in the introduction, two functions with unique characteristics are being compared – **`loop1`** and **`loop2`**. Each of these functions have 729 iterations in their outermost loops and are called a 100 times by the **`main`** function. The functions outermost loops are parallelised using orphaned directives and the loops which call the functions are within parallel regions. This minimises the overhead of starting and stopping the threads in every function call.
The scheduling strategy is set at set using the environment variable **`OMP_SCHEDULE`** and the runtime schedule in the code. The following results are obtained by executing on 6 threads (See Appendix B.1 for code and Appendix C.1 for PBS submission script).

1. Comparison of scheduling clauses for various chunk sizes

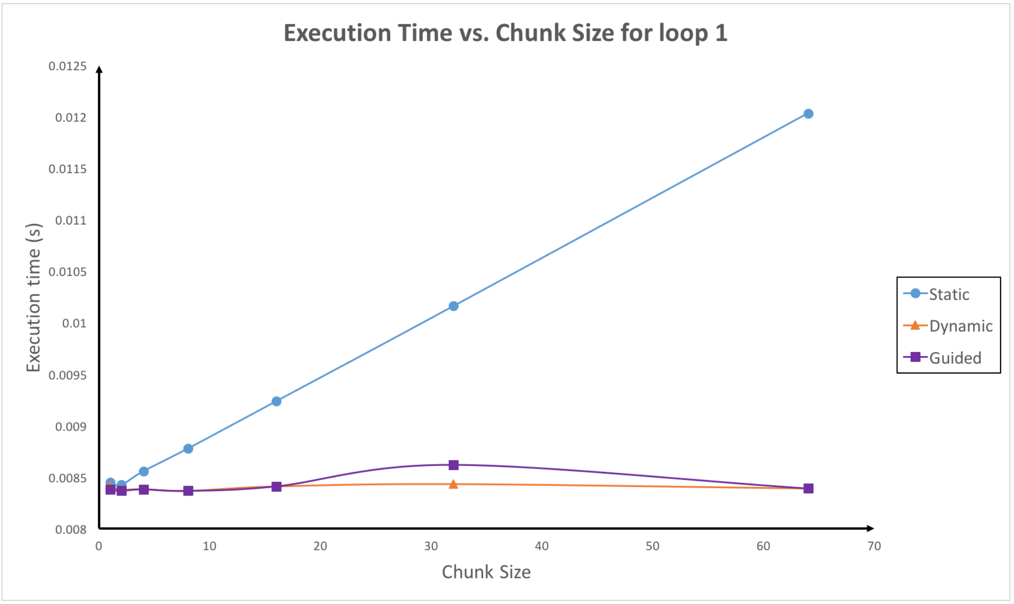

Figure 1: Graph of execution time of loop1 vs. chunk size for static, dynamic and guided schedules.

Figure 1 clearly illustrates that the execution time of **`loop1`** using static scheduling increases linearly with chunk size. Dynamic and guided schedules have roughly similar performances. Dynamic scheduling with a chunk size of 8 is the fastest strategy by only a small margin.

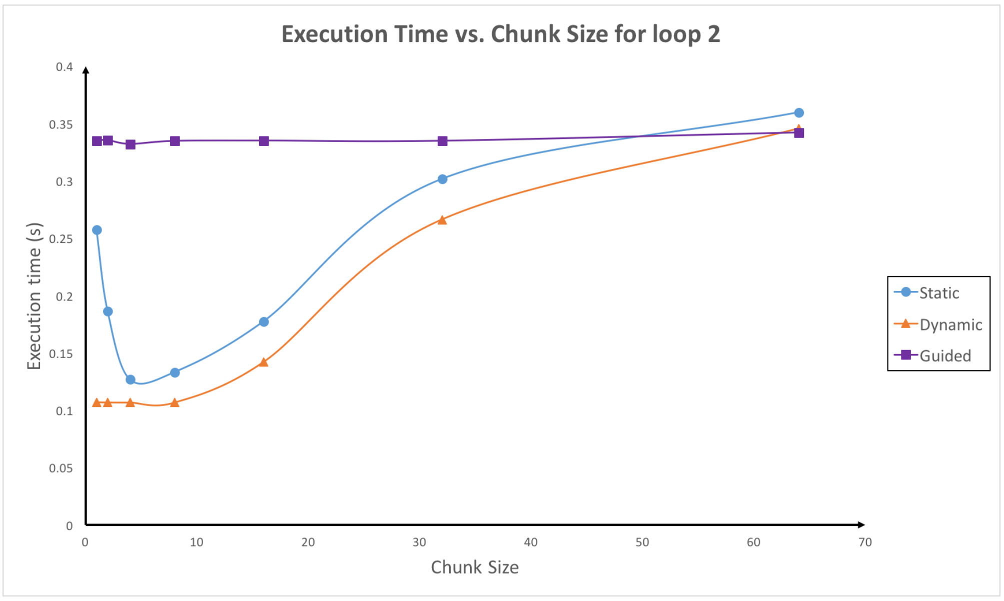

Figure 2: Graph of execution time of loop1 vs. chunk size for static, dynamic and guided schedules.

Figure 2 illustrates that the execution times of **`loop2`** using dynamic scheduling with a chunk size of 8 is the best scheduling strategy. Static scheduling with chunk size of 64 is the most costly strategy overall.

The performance trends observed above for **`loop1`** and **`loop2`** can be explained by considering the nature of iterations being carried out.

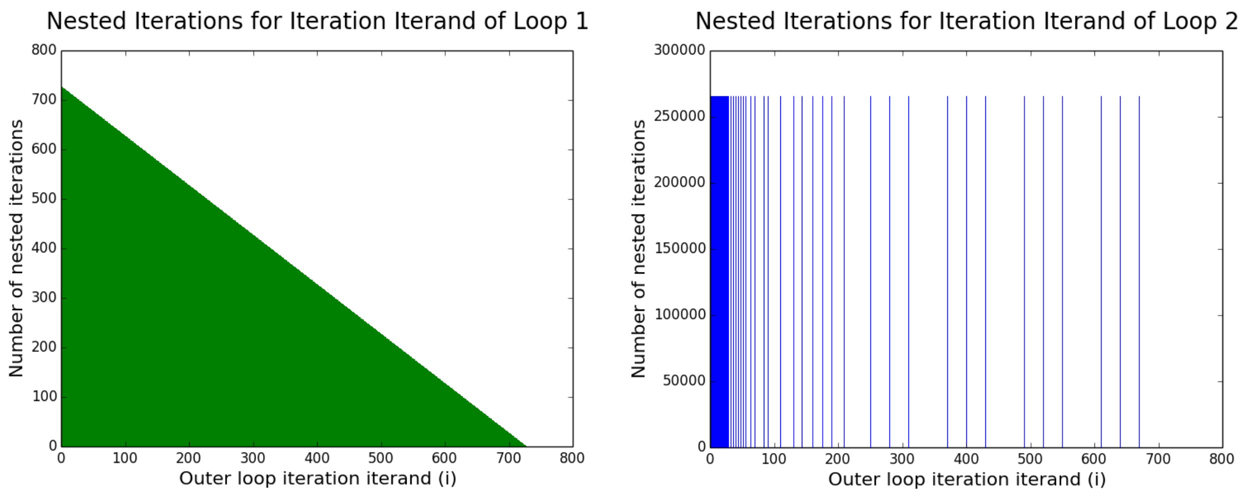

Figure 3: Bar graph of number of nested iterations to perform for iterand i for loop1 (left) and loop2 (right).

Both **`loop1`** and **`loop2`** have 729 iterations in their outer loop with an iterand **`i`**. Figure 3 is a thin bar graph of the number of iterations which have to be carried out in nested loops for each value of **`i`**.

In **`loop1`** the number of nested iterations linearly decreases from 728 to 0 with **`i`**. Loops for each iterand of **`loop1`** will greatly vary in execution times explaining why static scheduling is not the optimal approach for **`loop1`**. The load imbalance in static scheduling between adjacent chunks for **`loop1`** $=$ (chunk size)^2 (For derivation see Appendix A).

In **`loop2`**, the number of nested iterations are either 265356 or 0 depending upon the value of the expression:

`expr = i % (3 * (i/30) + 1)`

If **`expr == 0`**, 265356 have to be executed else no iterations have to be executed for that iterand **`i`**. Not all chunks will have an equal number of loaded iterands (iterands for which 265356 nested iterations have to be evaluated). The density of loaded iterands roughly decreases as **`i`** increases. As the chunk size increases the load imbalance increases. Thus guided scheduling is expensive for low minimum chunk sizes as well because the guided schedule starts with a large chunk size (121 in this case).

2. Variation of performance with number of threads

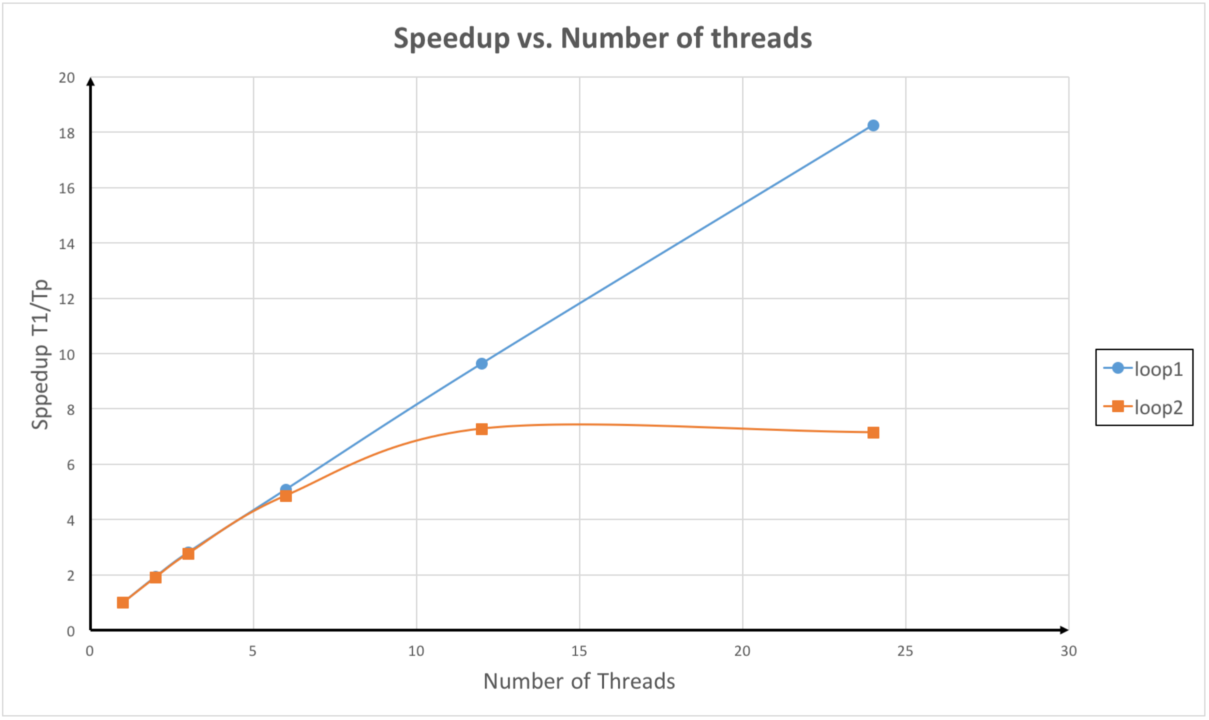

Figure 4: Speedup graph for loop1 and loop2. Schedule: Dynamic, 8

Figure 4 shows the speed up (execution time on 1 thread/ execution time on t threads) obtained by using the dynamic schedule clause with a chunk size of 8.

The performance improves linearly for **`loop1`**. For **`loop2`**, the speedup increases until it reaches a maximum value of 7.28 when 12 threads are used. Following this no further performance improvement is obtained (See Appendix C.2 for PBS submission script).

Affinity Scheduling

1. Assignment of local set

At the start of each function call, a thread is assigned a contiguous local set. To achieve this, the number of iterations that have to be distributed are divided by the number of threads available (**`P`**) to get the local set size – a shared object **`localSetSize`**. However, it is unlikely that **`P`** perfectly divides the number of iterations **`N`**. The default behaviour in C while dividing an integer by an integer is to round down the result to the next integer. The excess iterations are assigned to the thread with thread id (**`tid`**) = **`P - 1`** as seen in figure 5.

∴ **`localSetSizePm1 = N - (localSetSize * (P - 1))`**

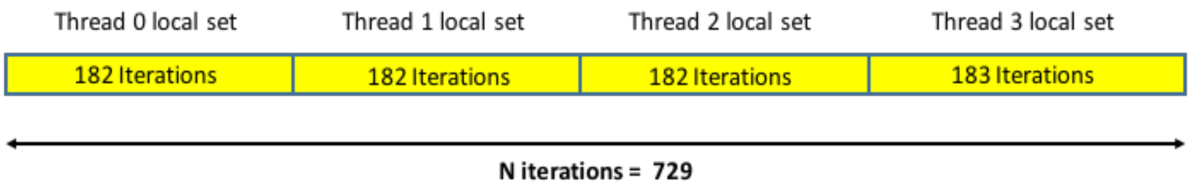

Figure 5: Distribution of iterations into local sets for 4 threads.

A thread determines its local set through its **`tid`** (= **`omp_get_thread_num()`**). Thus the lower bound of its local set (**`lb`**) = **`tid`** * **`localSetSize`**.

The thread then evaluates the loops starting from its lower bound in chunks of sizes **`chunkSize`** (= **`itrLeft`**/**`P`**). **`chunkSize`** is set to 1 when the iterations left in a thread's local set are less than **`P`**.

2. Load Transfer

The simple distribution of work as described in the previous sub-section requires no synchronisation as each thread works in its local set of iterations which is independent of the other threads. However, this results in the execution time being limited by the time taken to evaluate the largest local set and/or the slowest thread. Effectively the performance gain achieved is due to reduction of iterations required to be evaluated by a slow thread. When a thread completes the execution of its local set, it remains idle while waiting for the other 'loaded' threads to complete their local sets. By taking some iterations from the local set of loaded threads, theoretically the execution time can be reduced. This is the aim of affinity scheduling. The following objects are needed for load transfer.

`itrLeftArr`:

To make this possible, the number of iterations left in each threads local set should be accessible by other threads. For this an **`itrLeftArr`** array is needed. **`itrLeftArr`** is dynamically allocated enough bytes to store the **`itrLeft`** of each thread – it is an array of **`P`** integers. Thus when a thread has completed its local set, it iterates through the **`itrLeftArr`**, and finds the most loaded thread by comparing its values. The thread id of this most loaded thread is assigned to the object **`loadedT`**.

`localSetEnd`:

If the thread performing the load transfer finds a **`loadedT`**, it has to transfer a fraction of the iterations from **`loadedT`** to itself. The number of iterations transferred is assigned to the private variable **`itrLeft`** as follows

```
itrLeft = itrLeftArr[loadedT]/P
```

Following this the thread performing the load transfer has to update **`itrLeftArr`**, incrementing its own iterations by **`itrLeft`** and decreasing the iterations of **`loadedT`**, by the same amount.
However, the thread performing the load transfer does not know where to iterate from, as **`itrLeft`** has no indication of position in the **N** iterations. This necessitates the need for another object – shared **`localSetEnd`** array.
Like **`itrLeftArr`**, **`localSetEnd`** is an array of **`P`** integers representing the end position of the working set of each thread. As a rule, each time a thread acquires a set of iterations, it should update the value at its **`tid`** position in **`localSetEnd`** to provide this information to all the other threads. Additionally, after each load transfer a thread must update the value at the **`loadedT`** position of **`localSetEnd`** as well. The load transfer process is illustrated in figure 6.

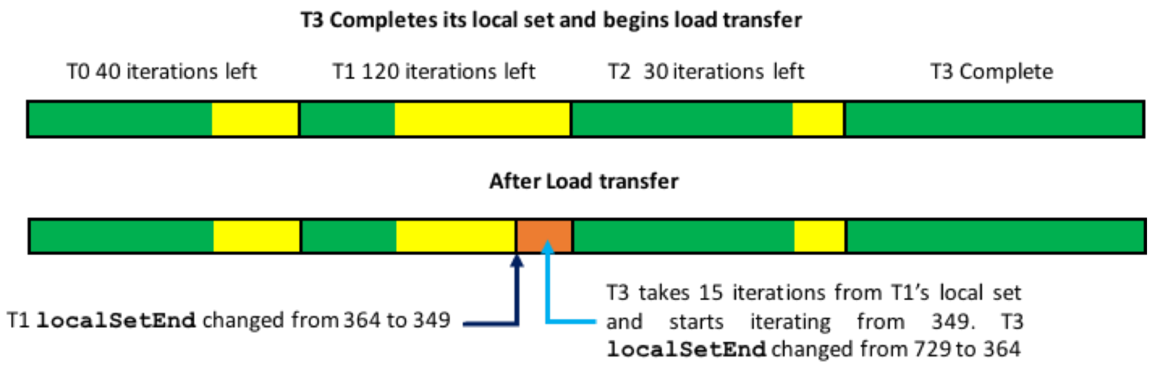
T3 Completes its local set and begins load transfer
T0 40 iterations left
T1 120 iterations left
T2 30 iterations left
T3 Complete
After Load transfer
T1 localSetEnd changed from 364 to 349
T3 takes 15 iterations from T1's local set and starts iterating from 349. T3 localSetEnd changed from 729 to 364

Figure 6: An illustrated example of a load transfer process carried out by thread 3 for 4 threads.

3. Synchronisation

Load transfer Synchronisation: Critical Section

The load transfer section of the code, which includes iterating over the **`itrLeftArr`** to find the most loaded thread and transferring iterations from the most loaded thread to itself, is enclosed in a critical section. This critical section is essential since if one thread updates the **`itrLeftArr`** while another thread (also in the load transfer section) is iterating over it, it will result in a race condition. Thus only one thread should be allowed to operate in the load transfer section at any given moment.

Thread data access synchronisation: Lock Array

While iterating over its local set a thread (let's call it t_i in this case) checks if it has iterations left in its local set, by reading **`itrLeftArr`**, before it evaluates a chunk. Following this, t_i decreases the iterations it has left in **`itrLeftArr`** by **`chunkSize`**. At the same time another thread (t_L) might perform load transfer from t_i's local set resulting in a race condition. Thus there is a possibility of t_L (thread performing the load transfer) racing with t_i (thread evaluating its local set) to update **`itrLeftArr`**. An atomic instruction is sufficient for reading **`itrLeftArr`** but is too limited to cover updates which involve load and store operations. Consider the operation:

```
#pragma omp atomic update
itrLeftArr[ti] -= chunkSize
```

In this case the atomic instruction will protect the assignment of the result of the expression to **`itrLeftArr[ti]`** but does not protect the evaluation of the expression [4]. A change in the value of **`itrLeftArr[ti]`**, during the evaluation, will almost certainly produce the wrong result.

Protecting **`itrLeftArr`** access with critical sections would be too restrictive as it will also prevent multiple threads, evaluating their respective local sets, from accessing and updating independent values in the **`itrLeftArr`** at the same time.

Using an array of locks (one lock for each thread's data) is the ideal solution as it do not cause the unnecessary slow down unlike critical sections and it is more versatile than atomic operations. **`lockThreadVal`** is a dynamically allocated array of **`P omp_lock_t`** objects. Any update to a certain position of shared array **`itrLeftArr`** requires acquiring the lock for that position. Additionally, thread t_i evaluating its local set acquires its lock before checking if it has iterations left. During this time no thread may perform a load transfer from t_i as the call to acquire a lock is blocking. This is illustrated in figure 7.

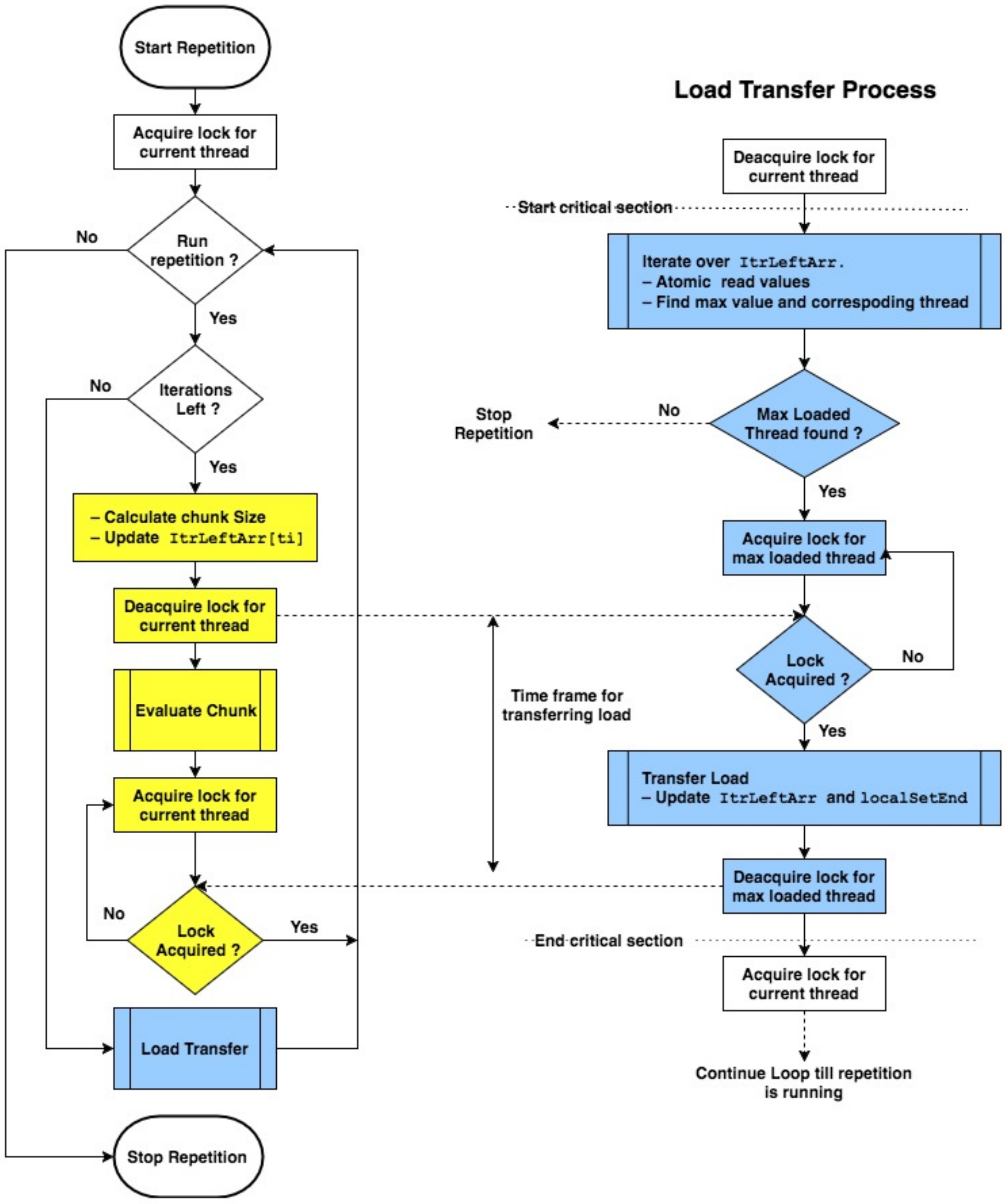
Start Repetition
Acquire lock for current thread
Run repetition ?
No
Yes
Iterations Left ?
No
Yes
– Calculate chunk Size
– Update ItrLeftArr[ti]
Deacquire lock for current thread
Evaluate Chunk
Acquire lock for current thread
Lock Acquired ?
No
Yes
Load Transfer
Stop Repetition
Load Transfer Process
Deacquire lock for current thread
Start critical section
Iterate over ItrLeftArr.
– Atomic read values
– Find max value and correspoding thread
Max Loaded Thread found ?
No
Stop Repetition
Yes
Acquire lock for max loaded thread
Lock Acquired ?
No
Yes
Time frame for transferring load
Transfer Load
– Update ItrLeftArr and localSetEnd
Deacquire lock for max loaded thread
End critical section
Acquire lock for current thread
Continue Loop till repetition is running

Figure 7: Process followed by each thread during each function call for a certain repetition (left) and the load transfer process elaborated (right).

Barrier at the end of each function call

Parallelised functions may be called multiple times within a loop for a certain number of repetitions. In each repetition, each thread has to perform a minimum of **`P – 1`** iterations as **`(P – 1)/P`** reduces to 0 and thus other threads cannot transfer iterations less than **`P`**. While a thread is evaluating these iterations in its own local set or another thread's local set (after load transfer), other threads may jump to the function call in the next repetition resulting in a race condition when those threads reach the load transfer. This will result in a race condition if two threads evaluate the same local set across two repetitions. Thus a barrier is needed at the end of each function call to ensure all threads are always operating in the function call in the same repetition. The occurrence of this race condition is illustrated in figure 8. As the number of threads increase, this race condition becomes far more likely and its effects becomes more apparent.

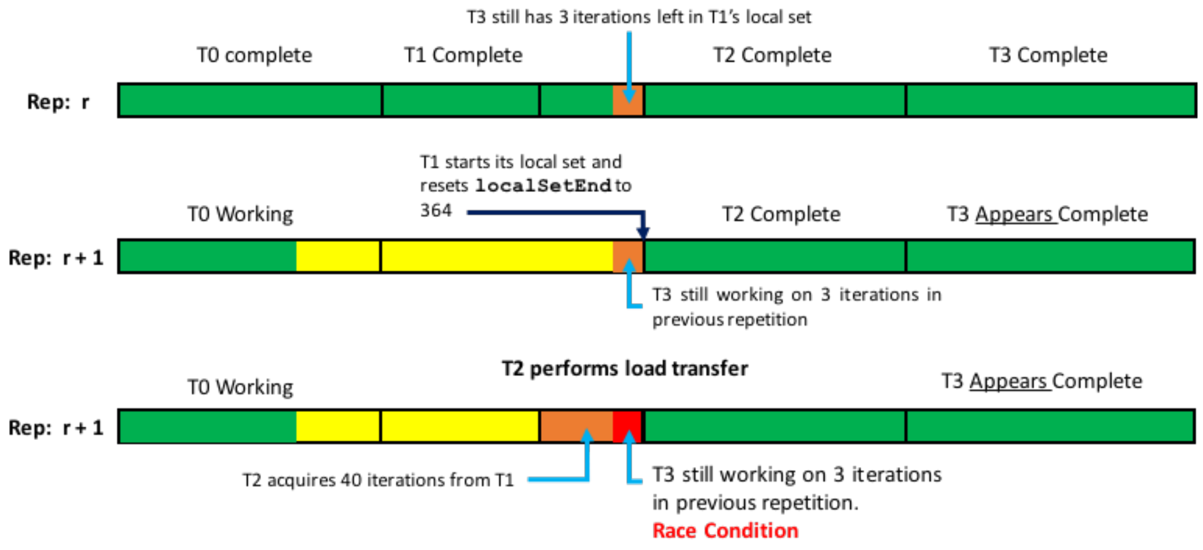
T3 still has 3 iterations left in T1's local set
T0 complete
T1 Complete
T2 Complete
T3 Complete
Rep: r
T1 starts its local set and resets localSetEnd to 364
T0 Working
T2 Complete
T3 Appears Complete
Rep: r + 1
T3 still working on 3 iterations in previous repetition
T2 performs load transfer
T0 Working
T3 Appears Complete
Rep: r + 1
T2 acquires 40 iterations from T1
T3 still working on 3 iterations in previous repetition.
Race Condition

Figure 8: Illustration of inter repetition race condition for 4 threads in the absence of a barrier at the end of the function call.

4. Performance

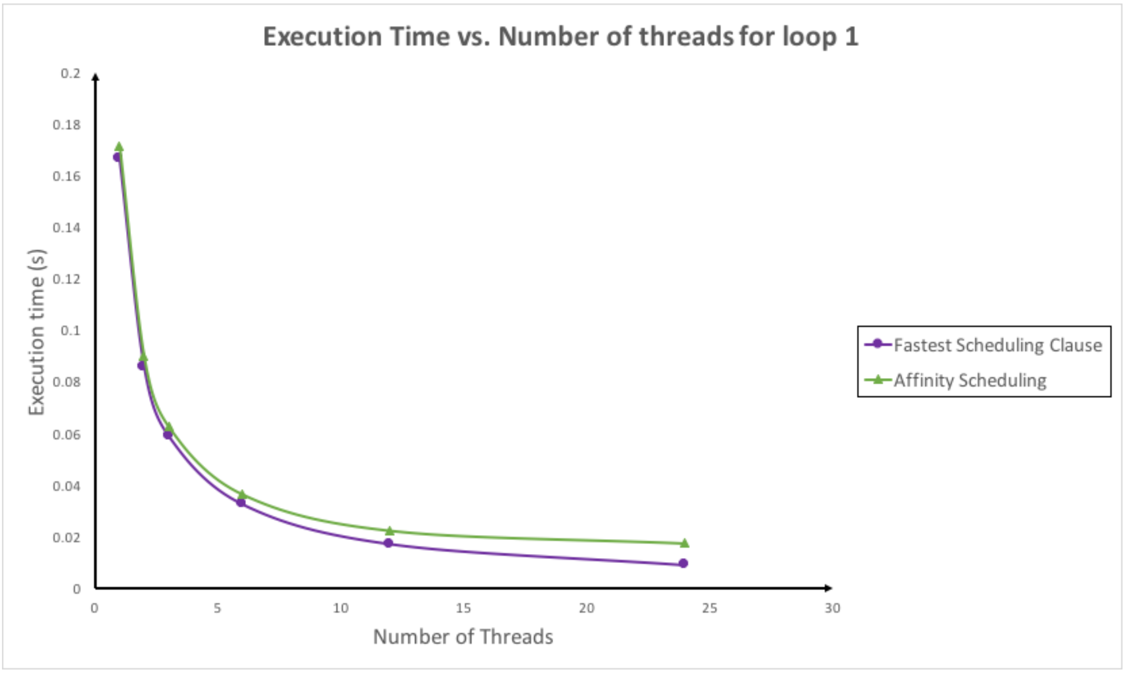

Figure 9: Loop1 Execution times of the Affinity scheduler and the fastest schedule clause:"dynamic, 8". Executed on 1,2,3,6,12 and 24 threads.

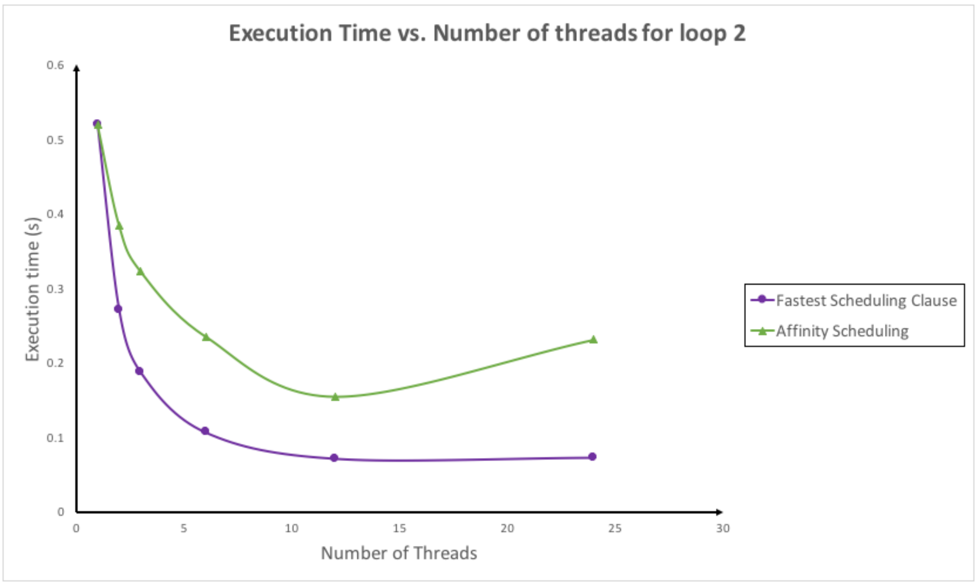

Figure 10: Loop2 Execution times of the Affinity scheduler and the fastest schedule clause:"dynamic, 8". Executed on 1,2,3,6,12 and 24 threads.

From figure 9 and figure 10, it is apparent that using a dynamic schedule with a chunk size of 8 would provide better performance than the designed affinity scheduler.
Table 1 and 2 shows the average time a spent by threads waiting at the critical section and the barrier of **`loop1`** and **`loop2`** in 100 function calls and all the load transfers (See Appendix B.2 for affinity scheduler source code C.3 for PBS submission script).

Table 1: Average time spent by a thread waiting for the load transfer critical section and the barrier in loop1.

Number of threads	Average idle time at Barrier		Average idle time at Load Transfer		Total execution time: t_e (s)
	(s)	% of t_e	(s)	% of t_e	
6	0.001743	4.9	0.000092	0.26	0.035584
12	0.003734	16.9	0.000165	0.75	0.022058
24	0.007013	39.4	0.000028	0.16	0.017818

Table 2: Average time spent by a thread waiting for the load transfer critical section and the barrier in loop2.

Number of threads	Average idle time at Barrier		Average idle time at Load Transfer		Total execution time: t_e (s)
	(s)	% of t_e	(s)	% of t_e	
6	0.133322	56.8	0.000078	0.03	0.234648
12	0.104844	66.0	0.000172	0.11	0.158968
24	0.208213	87.64	0.000102	0.04	0.237585

From the above data it is clear that the major source of overhead is the barrier which accounted for approximately 88% of the cost when parallelising **`loop2`** with 24 threads using the affinity scheduler.

Conclusion

The dynamic schedule clause with a chunk size of 8 was found to be the best scheduling strategy among the scheduling options available in OpenMP for the given program. Dynamic scheduling is the ideal approach while dealing with loops in which execution times of iterations vary greatly. The reduction in load imbalance more than compensates for the overhead of acquiring a fresh chunk of iterations. Also smaller chunk sizes further mitigate the potential for load imbalance in such a situation. However smaller chunk sizes result in lower data affinity. Hence a chunk size of 8 was found to be the ideal chunk size for minimising the total execution time.

The developed affinity scheduler was not faster than the dynamic scheduler (chunk size 8). The primary source of overhead was found to be the barrier at the end of the function. Since there is an implicit barrier at the end of the OpenMP for work-sharing construct, it would be safe to conclude that the idle threads at the barrier is the source of overhead for the dynamic scheduler as well.

As the number of threads increase, the minimum number of iterations a thread has to perform in the affinity scheduled program, described in this report, increases as well (23 iterations when run on 24 threads). This has the same effect as using large chunk sizes with OpenMP scheduling directives which in turn increases overheads.

References

[1] J.M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", EPCC, University of Edinburgh, Edinburgh, 2017.

[2] E.A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP", in *Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*, 1989.

[3] Tao Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, 1994.

[4] "Pragma directives for parallel processing", *Ibm.com*, 2017. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSGH2K_11.1.0/com.ibm.xlc111.aix.doc/compiler_ref/prag_omp_atomic.html. [Accessed: 14th April 2017].

Appendix

Appendix A: Load imbalance between adjacent chunks for loop1

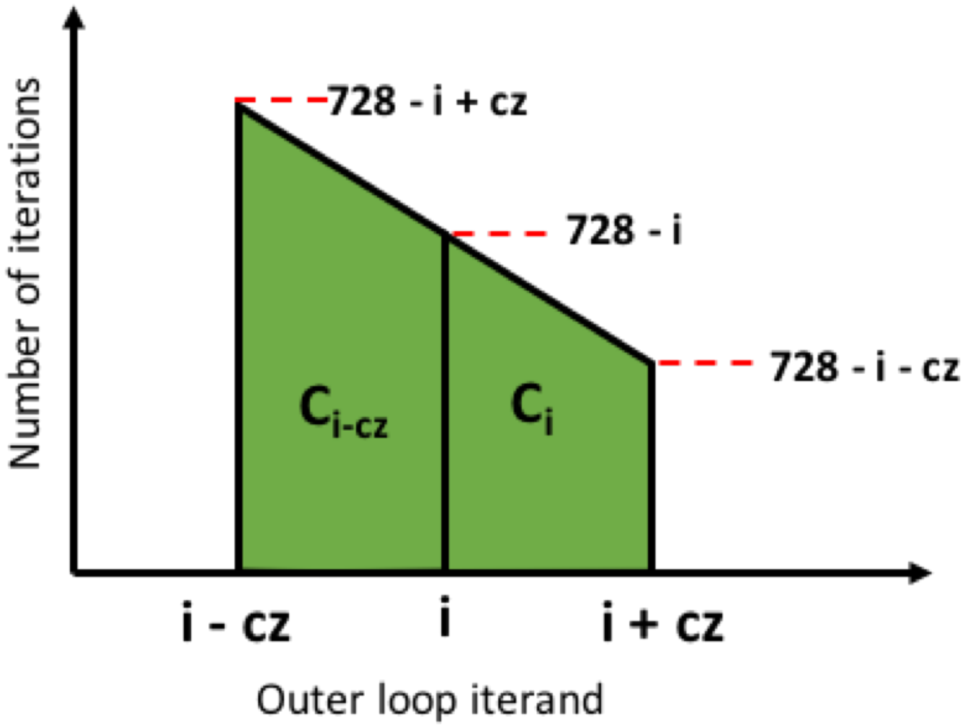

Figure 11: Graphical representation of number of iterations in adjacent chunks of iterations for loop1

The total number of iterations in the chunks shown in figure 11 can be calculated as areas of right-angled trapezoids.

$$C_{i-cz} = cz \cdot \left(728 - i + \frac{cz}{2}\right) \tag{1}$$

$$C_i = cz \cdot \left(728 - i - \frac{cz}{2}\right) \tag{2}$$

where:
i = iterand
cz = chunk size
C_i = area of chunk starting at i^{th} iterand

subtracting equation (2) from (1) we get the load imbalance between the chunks:

$$load\ imbalance = C_{i-cz} - C_i = cz^2 \tag{3}$$

Appendix B: C scripts

Available at: https://github.com/RoshanPasupathy/OMP-coursework

B.1 Comparing Scheduling Clauses – loopsSchedule.c

```c
#include <stdio.h>
#include <math.h>


#define N 729
#define reps 100
#include <omp.h>

double a[N][N], b[N][N], c[N];
int jmax[N];
int t_no;

void init1(void);
void init2(void);
void loop1(void);
void loop2(void);
void valid1(void);
void valid2(void);

int main(int argc, char *argv[]) {

  double start1,start2,end1,end2;
  int r;
  init1();

  start1 = omp_get_wtime();
  #pragma omp parallel default(none) \
  shared(a , b) \
  private(r)
  {
  for (r=0; r<reps; r++){
    loop1();
  }
  }
  end1  = omp_get_wtime();

  valid1();

  printf("Total time for %d reps of loop 1 = %f\n",reps, (float)(end1-start1));


  init2();

  start2 = omp_get_wtime();

  #pragma omp parallel default(none) \
  shared(a , b) \
  private(r)
  {
  for (r=0; r<reps; r++){
    loop2();
  }
  }

  end2  = omp_get_wtime();

  valid2();

  printf("Total time for %d reps of loop 2 = %f\n",reps, (float)(end2-start2));

```

```
}

void init1(void){
  int i,j;

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      a[i][j] = 0.0;
      b[i][j] = 3.142*(i+j);
    }
  }

}

void init2(void){
  int i,j, expr;

  for (i=0; i<N; i++){
    expr =  i%( 3*(i/30) + 1);
    if ( expr == 0) {
      jmax[i] = N;
    }
    else {
      jmax[i] = 1;
    }
    c[i] = 0.0;
  }

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      b[i][j] = (double) (i*j+1) / (double) (N*N);
    }
  }

}

void loop1(void) {
  int i,j;
  // dynamic, 8 best schedule option
  #pragma omp for schedule(runtime)
  for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }
}



void loop2(void) {
  int i,j,k;
  double rN2;
  rN2 = 1.0 / (double) (N*N);
  // dynamic, 8 best schedule option
  #pragma omp for schedule(runtime)
  for (i=0; i<N; i++){
    for (j=0; j < jmax[i]; j++){
      for (k=0; k<j; k++){
    c[i] += (k+1) * log (b[i][j]) * rN2;
      }
    }
  }
}

void valid1(void) {
  int i,j;
  double suma;

  suma= 0.0;
  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      suma += a[i][j];
    }
  }
  printf("Loop 1 check: Sum of a is %lf\n", suma);

}


void valid2(void) {
  int i;
  double sumc;

  sumc= 0.0;
  for (i=0; i<N; i++){
    sumc += c[i];
  }
  printf("Loop 2 check: Sum of c is %f\n", sumc);
}
```

B.2 Affinity Scheduler – loopsAffinity.c

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


#define N 729
#define reps 100
#include <omp.h>

double a[N][N], b[N][N], c[N];
int jmax[N];

int cnt_ldd_ts[reps]; //number of loaded threads per rep
int P; //number of threads
int tf; //transfer factor
int seg_sz; //iterations allocated to each thread except last
int seg_szP;
int* itrLeftArr; //iterations left in each thread
int* d_seg_sz; // seg assigned to each thread
int* localSetEnd; // end of segments
int* ldd_tarr; //array of loaded threads
int* default_ldd_tarr;
int* getThread;
double* idleTimeBarrier;
double* idleTimeLT;

int* thread_rng; //indicate thread is running
int* is_assigned;

int ut;
omp_lock_t lock_ut;
int r; //reps iterator
//int tot_itr = N; //total iterations left
//int t_no;
//int* itr_prf;
int itr_prf[N] = {0};
int max_threads;
omp_lock_t* lock_itrLeftArr;

void init1(void);
void init2(void);
void loop1(void);
void loop2(void);
void valid1(void);
void valid2(void);
void debug_race(void);

int main(int argc, char *argv[]) {
  //int r;
  double start1,start2,end1,end2;
  int t, lk, i;
  double idletimeb;
  double idletimel;

  max_threads = omp_get_max_threads();

  P = max_threads;

  //tf = P/(float) (P - 1);

  //Constant values
  tf = P;
  printf("number of threads: %d\n", P);
  for (i =0; i < reps; i++){
    cnt_ldd_ts[i] = P;
  }
  ut = P;
  seg_sz = (int) (N/P);
  seg_szP = seg_sz + N - (seg_sz * P);
  lock_itrLeftArr = (omp_lock_t*)malloc(sizeof(omp_lock_t) * (P + 1));

  //Variable values
  idleTimeBarrier = (double *) malloc(sizeof(double) * P);
  idleTimeLT = (double *) malloc(sizeof(double) * P);
  d_seg_sz = (int *) malloc(sizeof(int) * P);
  itrLeftArr = (int *) malloc(sizeof(int) * P);
  localSetEnd = (int *) malloc(sizeof(int) * P);
  ldd_tarr = (int *) malloc(sizeof(int) * P);
  default_ldd_tarr = (int *) malloc(sizeof(int) * P);
  thread_rng = (int *) malloc(sizeof(int) * P);
  is_assigned = (int *) malloc(sizeof(int) * P);

  if ((itrLeftArr == NULL) || (d_seg_sz == NULL) || (lock_itrLeftArr == NULL) || (localSetEnd
== NULL) ) {
    printf("FATAL: Malloc failed ... \n");
  }
  else{
    printf("Allocation: OKAY\n");
  }
  //Initialise lock
  omp_init_lock(&lock_ut);
  for (lk = 0; lk < P; lk++){
    idleTimeBarrier[lk] = 0;
    idleTimeLT[lk] = 0;
    omp_init_lock(&(lock_itrLeftArr[lk]));
    itrLeftArr[lk] = 0;
    d_seg_sz[lk] = seg_sz;
    ldd_tarr[lk] = lk; //at the start all threads are loaded
    default_ldd_tarr[lk] = lk;
    thread_rng[lk] = 0;
    localSetEnd[lk] = (lk + 1) * seg_sz;
    is_assigned[lk] = 0;
  }
  getThread = default_ldd_tarr;
  localSetEnd[P-1] = N;
  is_assigned[P] = 0;
  d_seg_sz[P-1]+= N - (seg_sz*P);
  //localSetEnd[P-1] = N;

  init1();
  //printf("iterations allocated to each thread: %d\n", seg_sz);

  start1 = omp_get_wtime();

  #pragma omp parallel default(none) \
  shared(P, tf, seg_sz,seg_szP, itrLeftArr,d_seg_sz, localSetEnd) \
  shared(idleTimeLT, idleTimeBarrier, lock_ut, thread_rng, is_assigned, default_ldd_tarr, getT
hread) private(r)
  {
  for (r=0; r<reps; r++){
    //#pragma omp critical(printf_lock)
    //{printf("main function r: %d\n",r);}
    loop1();
  // }
  }
  }
  end1  = omp_get_wtime();

  valid1();
```

```c

  printf("Total time for %d reps of loop 1 = %f\n",reps, (float)(end1-start1));
  for (i =0; i < reps; i++){
    cnt_ldd_ts[i] = P;
  }
  getThread = default_ldd_tarr;
  idletimeb = 0;
  idletimel = 0;
  //printf("idle time 1 = %d\n", idleTimeLT[0]);
  for (t = 0; t < P; t++){
    thread_rng[t] = 0;
    idletimeb += idleTimeBarrier[t];
    idletimel += idleTimeLT[t];
    idleTimeBarrier[t] = 0;
    idleTimeLT[t] = 0;
  }
  printf("Loop 1: Average idle time at load transfer = %f , Average idle time at barrier = %f\n",idletimel/(double) P, idletimeb/(double) P);
  init2();
  start2 = omp_get_wtime();
  #pragma omp parallel default(none) \
  shared(P, tf, seg_sz,seg_szP, itrLeftArr,d_seg_sz, localSetEnd) \
  shared(idleTimeBarrier, idleTimeLT, ut, lock_ut, thread_rng, is_assigned, default_ldd_tarr, getThread) private(r)
  {
  for (r=0; r<reps; r++){
    loop2();
  }
  }
  end2  = omp_get_wtime();

  valid2();

  printf("Total time for %d reps of loop 2 = %f\n",reps, (float)(end2-start2));
  idletimeb = 0;
  idletimel = 0;
  //printf("idle time 1 = %d\n", idleTimeLT[0]);
  for (t = 0; t < P; t++){
    thread_rng[t] = 0;
    idletimeb += idleTimeBarrier[t];
    idletimel += idleTimeLT[t];
    idleTimeBarrier[t] = 0;
    idleTimeLT[t] = 0;
  }
  printf("Loop 2: Average idle time at load transfer = %f , Average idle time at barrier = %f\n",idletimel/(double) P, idletimeb/(double) P);
  //debug_race();
  free(d_seg_sz);
  free(itrLeftArr);
  free(lock_itrLeftArr);
  free(localSetEnd);
  free(ldd_tarr);
  free(default_ldd_tarr);
  free(thread_rng);
}

void init1(void){
  int i,j;

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      a[i][j] = 0.0;
      b[i][j] = 3.142*(i+j);
    }
  }

```

```
}

void init2(void){
  int i,j, expr;

  for (i=0; i<N; i++){
    expr =  i%( 3*(i/30) + 1);
    if ( expr == 0) {
      jmax[i] = N;
    }
    else {
      jmax[i] = 1;
    }
    c[i] = 0.0;
  }

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      b[i][j] = (double) (i*j+1) / (double) (N*N);
    }
  }

}

void loop1(void) {
  int i,j,t, lb;
  int chnk_sz, ub;
  //maximum load, load, loaded thread
  int max_val, val, ldd_t;
  int itr, t_no;
  double tstart, tend;
  /*
  default initial segment is
  corresponds to thread number
  */
  t_no = omp_get_thread_num();
  //set iterations
  max_val = tf;
  lb = t_no * seg_sz;
  omp_set_lock(&(lock_itrLeftArr[t_no]));
  itr = (t_no == P - 1)? seg_szP: seg_sz;
  itrLeftArr[t_no] = itr;
  localSetEnd[t_no] = (t_no * seg_sz) + itr;
  while (max_val > 0){
    while (itr > 0){
      chnk_sz = (itr < P)? 1: (int) (itr/P);
      itrLeftArr[t_no] -= chnk_sz;
      omp_unset_lock(&(lock_itrLeftArr[t_no]));
      ub = lb + chnk_sz;
      for (;lb < ub; lb++){
        //pragma omp atomic update //DEBUG
        //itr_prf[lb]++;             //DEBUG
        for (j=N-1; j>lb; j--){
          a[lb][j] += cos(b[lb][j]);
        }
      }
      omp_set_lock(&(lock_itrLeftArr[t_no]));
      itr = itrLeftArr[t_no];
      // New chunk size
    } // Ran out of iterations in assigned segment
    //indicate to all threads that one less thread is running in seg
    omp_unset_lock(&(lock_itrLeftArr[t_no]));

   // Load transfer block //
    max_val = tf;
    ldd_t = t_no;
    tstart = omp_get_wtime();
    #pragma omp critical (load_transfer)
    {
    tend = omp_get_wtime();
    idleTimeLT[t_no] += tend - tstart;
    //#pragma omp critical (printf_lock)
    //{printf("Time spent at critical = %f\n",tend=tstart);}
    for (t = 0; t < P; t++){
      #pragma omp atomic read
      val = itrLeftArr[t];
      if (val > max_val){
        max_val = val;
        ldd_t = t;
      }
    }
    //transfer load if new thread found else do nothing ?
    if (ldd_t != t_no){
      omp_set_lock(&(lock_itrLeftArr[ldd_t]));
      itr = (int) (itrLeftArr[ldd_t]/tf);
      itrLeftArr[ldd_t] -= itr;
      omp_unset_lock(&(lock_itrLeftArr[ldd_t]));
      localSetEnd[t_no] = localSetEnd[ldd_t];
      localSetEnd[ldd_t] -= itr;
      itrLeftArr[t_no] = itr;
      // Get starting value
      #pragma omp atomic read
      lb = localSetEnd[ldd_t];
    }
    } //END LOAD TRANSFER
    if (ldd_t == t_no) {
      max_val = 0;
    }

    else {
      omp_set_lock(&(lock_itrLeftArr[t_no]));
    }
  }
  tstart = omp_get_wtime();
  #pragma omp barrier
  tend = omp_get_wtime();
  idleTimeBarrier[t_no] += tend - tstart;
}



void loop2(void) {
  int i,j,k;
  double rN2 = 1.0 / (double) (N*N);;
  int t, lb;
  int chnk_sz, ub;
  //maximum load, load, loaded thread
  int max_val, val, ldd_t;
  int itr, t_no;
  double tstart,tend;
  /*
  default initial segment is
  corresponds to thread number
  */
  t_no = omp_get_thread_num();
  //set iterations
  max_val = tf;
  lb = t_no * seg_sz;

  omp_set_lock(&(lock_itrLeftArr[t_no]));
  itr = (t_no == P - 1)? seg_szP: seg_sz;
  itrLeftArr[t_no] = itr;
```

```
localSetEnd[t_no] = (t_no * seg_sz) + itr;
while (max_val > 0){
  while (itr > 0){
    chnk_sz = (itr < P)? 1: (int) (itr/P);
    itrLeftArr[t_no] -= chnk_sz;
    omp_unset_lock(&(lock_itrLeftArr[t_no]));
    ub = lb + chnk_sz;
    for (; lb<ub; lb++){
      for (j=0; j < jmax[lb]; j++){
        for (k=0; k<j; k++){
          c[lb] += (k+1) * log (b[lb][j]) * rN2;
        }
      }
    }
    omp_set_lock(&(lock_itrLeftArr[t_no]));
    itr = itrLeftArr[t_no];
    // New chunk size
  } // Ran out of iterations in assigned segment

  //indicate to all threads that one less thread is running in seg
  omp_unset_lock(&(lock_itrLeftArr[t_no]));
 // Load transfer block //
  max_val = tf;
  ldd_t = t_no;
  tstart = omp_get_wtime();
  #pragma omp critical (load_transfer)
  {
  tend = omp_get_wtime();
  idleTimeLT[t_no] += tend - tstart;
  for (t = 0; t < P; t++){
    #pragma omp atomic read
    val = itrLeftArr[t];
    if (val > max_val){
      max_val = val;
      ldd_t = t;
    }
  }
  //transfer load if new thread found else do nothing ?
  if (ldd_t != t_no){
    omp_set_lock(&(lock_itrLeftArr[ldd_t]));
    itr = (int) (itrLeftArr[ldd_t]/tf);
    itrLeftArr[ldd_t] -= itr;
    omp_unset_lock(&(lock_itrLeftArr[ldd_t]));
    localSetEnd[t_no] = localSetEnd[ldd_t];
    localSetEnd[ldd_t] -= itr;
    itrLeftArr[t_no] = itr;
    // Get starting value
    #pragma omp atomic read
    lb = localSetEnd[ldd_t];
  }
  } //END LOAD TRANSFER
  if (ldd_t == t_no) {
    max_val = 0;
  }
  else{
    omp_set_lock(&(lock_itrLeftArr[t_no]));
  }
}
tstart = omp_get_wtime();
#pragma omp barrier
tend = omp_get_wtime();
idleTimeBarrier[t_no] += tend - tstart;
}



```

```c
void valid1(void) {
  int i,j;
  double suma;

  suma= 0.0;
  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      suma += a[i][j];
    }
  }
  printf("Loop 1 check: Sum of a is %lf\n", suma);

}


void valid2(void) {
  int i;
  double sumc;

  sumc= 0.0;
  for (i=0; i<N; i++){
    sumc += c[i];
  }
  printf("Loop 2 check: Sum of c is %f\n", sumc);
}

void debug_race(void){
  int i,seg, ub;
  printf("running debug... \n");
  for (seg = 0; seg < P; seg++){
    printf("Segment %d... \n", seg);
    ub = (seg_sz + 1) * seg;
    if (seg == P - 1){
      ub = N;
    }
    for (i = seg_sz * seg; i < ub ; i++){
      if (itr_prf[i] > reps){
        printf("At %d th value a race is occuring, excess it: %d\n", i, itr_prf[i] - reps);
      }
      else if (itr_prf[i] < reps){
        printf("At %d th value insufficient iterations have been executed, lacking: %d\n", i,
reps-itr_prf[i]);
      }
    }
    printf("\n");
  }
}
```

Appendix C: PBS submission script

Available at: https://github.com/RoshanPasupathy/OMP-coursework

C.1 Scheduling Clauses – scheduleComp.pbs

```
#
# All screen output (stdout and stderr) will appear in a file called
# myjob.pbs.oXXXXX, where XXXXX is the job number assigned at submit time.
#
# Roshan Pasupathy, rp3g13
# based on code by David Henty, EPCC, 25/11/2014
#
# Note: Due to timeout, this PBS submission
#       script was submitted in three parts
#       >> scheduleComp1.pbs ran chunk sizes {1,2,4}
#       >> scheduleComp2.pbs ran chunk sizes {8,16}
#       >> scheduleComp3.pbs ran chunk sizes {32,54}

#PBS -A y14-CDT-Soton
#PBS -j oe
#PBS -l walltime=00:01:00
#PBS -l select=1

cd $PBS_O_WORKDIR

echo '--------------------------------------------------------------------------------'
export OMP_NUM_THREADS=6

echo 'Started at' `date`
echo '--------------------------------------------------------------------------------'
echo 'NUMBER OF THREADS: ' $OMP_NUM_THREADS
echo.

for chunksize in {1,2,4,8,16,32,64}; do
    for sched in {"static","dynamic","guided"}; do
        echo '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        export OMP_SCHEDULE="${sched},${chunksize}"
        echo 'SCHEDULE: ' $sched 'CHUNKSIZE: ' $chunksize
        echo '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        (time aprun -n 1 -d $OMP_NUM_THREADS ./loopsSchedule) 2>&1
        echo '--------------------------------------------------------------------------------'
        echo.
    done
done

echo '--------------------------------------------------------------------------------'
echo 'Finished at' `date`
~
```

C.2 Speedup vs. Variation in number of Threads – speedThreads.pbs

```
#
# All screen output (stdout and stderr) will appear in a file called
# myjob.pbs.oXXXXX, where XXXXX is the job number assigned at submit time.
#
# Roshan Pasupathy, rp3g13
# based on code by David Henty, EPCC, 25/11/2014
#
# Note: Due to timeout, this PBS submission
#       script was submitted in three parts
#       >> speedThreads1.pbs ran chunk sizes {1,2}
#       >> speedThreads2.pbs ran chunk sizes {3,6}
#       >> speedThreads3.pbs ran chunk sizes {12,24}

#PBS -A y14-CDT-Soton
#PBS -j oe
#PBS -l walltime=00:01:00
#PBS -l select=1

cd $PBS_O_WORKDIR

echo '---------------------------------------------------------------------------------'


echo 'Started at' `date`
echo '---------------------------------------------------------------------------------'

export OMP_SCHEDULE="dynamic,8"

for threads in {1,2,3,6,12,24}; do
    for script in {"loopsSchedule"}; do
        export OMP_NUM_THREADS=$threads
        echo '+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        echo 'Running' $script 'on'  $OMP_NUM_THREADS 'threads'
        echo '+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        (time aprun -n 1 -d $OMP_NUM_THREADS ./$script) 2>&1
        echo '---------------------------------------------------------------------------------'
    done
done

echo '---------------------------------------------------------------------------------'
echo 'Finished at' `date`
~
```

C.3 Affinity Scheduler Performance – exectimeAffinity.pbs

```
#
# All screen output (stdout and stderr) will appear in a file called
# myjob.pbs.oXXXXX, where XXXXX is the job number assigned at submit time.
#
# Roshan Pasupathy, rp3g13
# based on code by David Henty, EPCC, 25/11/2014
#
# Note: Due to timeout, this PBS submission
#       script was submitted in three parts
#       >> exectimeAffinity1.pbs ran chunk sizes {1,2}
#       >> exectimeAffinity 2.pbs ran chunk sizes {3,6}
#       >> exectimeAffinity 3.pbs ran chunk sizes {12,24}

#PBS -A y14-CDT-Soton
#PBS -j oe
#PBS -l walltime=00:01:00
#PBS -l select=1

cd $PBS_O_WORKDIR

echo '--------------------------------------------------------------------------------'


echo 'Started at' `date`
echo '--------------------------------------------------------------------------------'

export OMP_SCHEDULE="dynamic,8"

for threads in {1,2,3,6,12,24}; do
    for script in {"loopsSchedule","loopsAffinity"}; do
        export OMP_NUM_THREADS=$threads
        echo '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        echo 'Running' $script 'on'  $OMP_NUM_THREADS 'threads'
        echo '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++'
        (time aprun -n 1 -d $OMP_NUM_THREADS ./$script) 2>&1
        echo '--------------------------------------------------------------------------------'
    done
done

echo '--------------------------------------------------------------------------------'
echo 'Finished at' `date`
~
```

Appendix D: Makefile

Available at: https://github.com/RoshanPasupathy/OMP-coursework

```makefile
# Makefile for the FEEG6003 coursework

#
# C compiler and options for system
#
CC= cc
CFLAGS=-O3
LIB= -lm

SRCS=$(wildcard loop*.c)
EXCS=$(patsubst %.c,%,$(SRCS))
all: $(EXCS)

#
# Compile
#
% : %.c
	$(CC) $(CFLAGS) -o $@ $< $(LIB)
#
# Clean out object files and the executable.
#
clean:
	rm $(EXCS)
```