# Affinity Scheduling

## 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 1.

∴ **localSetSizePm1** = **N** – (**localSetSize** \* (**P** – 1))



Figure 1: 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 threads local set are less than **P**.

## 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.

### Necessary Objects

**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 i.e. 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 the iterations left in the local sets. 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, if a thread transfers load from another **loadedT**, it must update the value at the **loadedT** position of **localSetEnd** as well. The load transfer process is illustrated in figure 2.

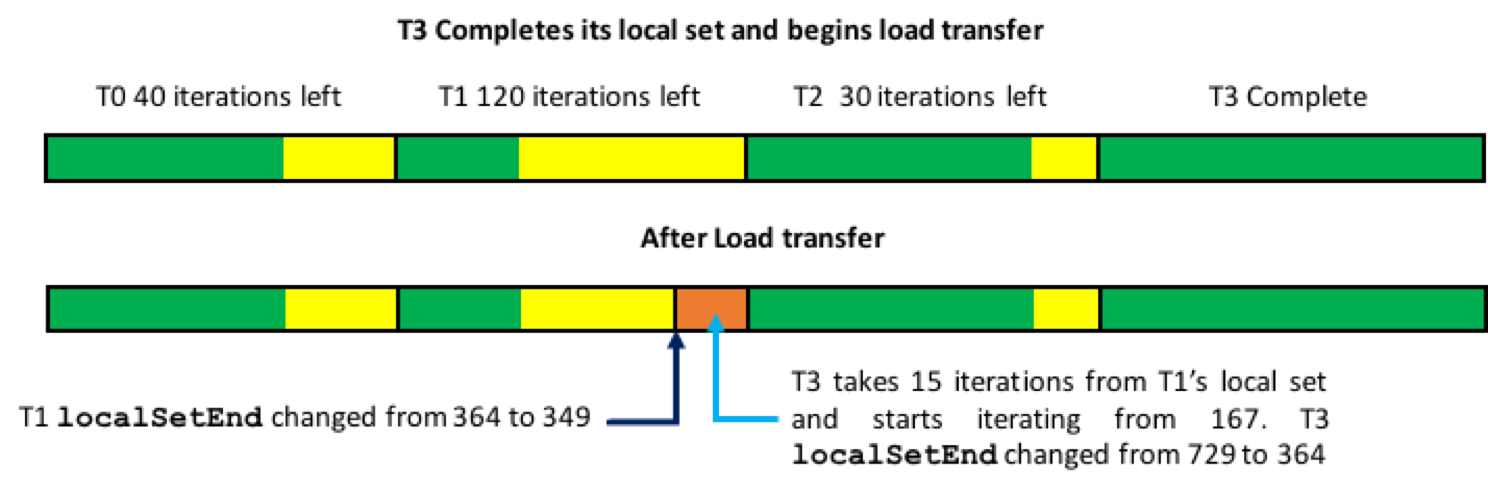


Figure 2: An illustrated example of a load transfer process carried out by thread 3 for 4 threads.

### 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 ti in this case) checks if it has iterations left in its local set, by reading **itrLeftArr**, before it evaluates a chunk. Following this, ti decreases the iterations it has left in **itrLeftArr** by **chunkSize**. At the same time another thread (tL) might perform load transfer from ti’s local set resulting in a race condition. Thus there is a possibility of tL (thread performing the load transfer) racing with ti (thread evaluating its local set) racing 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 **itrLeftArr[ti]** − **chunkSize** to **itrLeftArr[ti]** but does not protect the evaluation of the expression [1]. A change in the value of **itrLeftArr[ti]**, will almost certainly produce the wrong result.

[1] <https://www.ibm.com/support/knowledgecenter/SSGH2K_11.1.0/com.ibm.xlc111.aix.doc/compiler_ref/prag_omp_atomic.html>

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 arrays (**itrLeftArr**) requires acquiring the lock for that position. Additionally, thread ti 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 ti as the call to acquire a lock is blocking. This is illustrated in figure 3

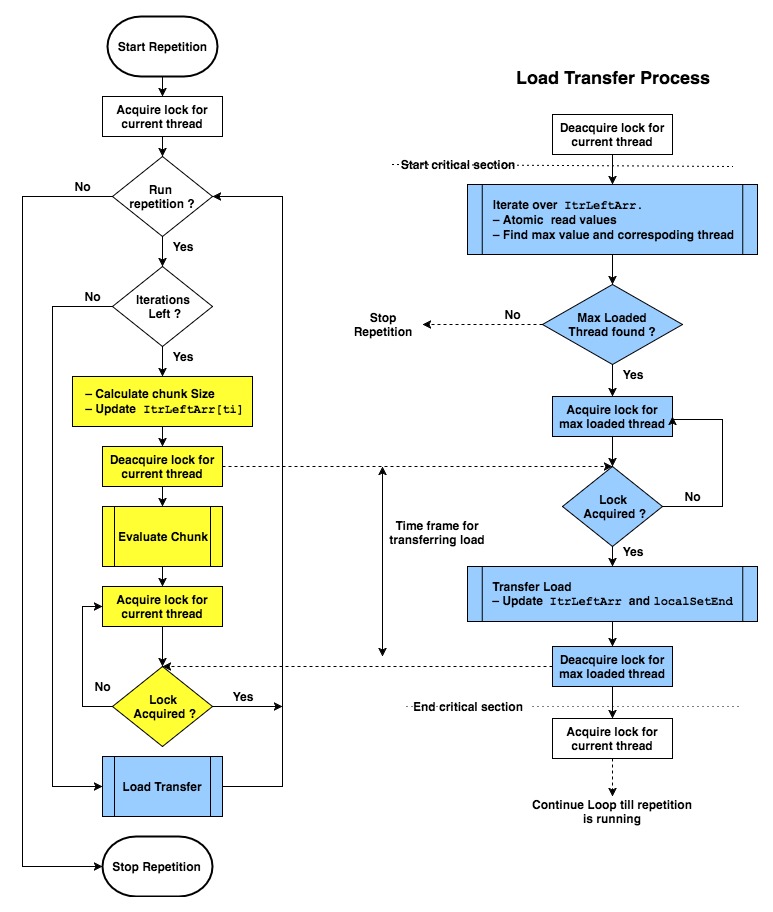
****

Figure 3: 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 4. As the number of threads increase, this race condition becomes far more likely and its effects becomes more apparent.



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