Introduction to SWARM
Software and Algorithms for Running on Multicore Processors

David A. Bader
Georgia Institute of Technology
http://www.cc.gatech.edu/~bader

Tutorial compiled by
Rucheek H. Sangani
M.S. Student, College of Computing
Georgia institute of Technology
http://www.cc.gatech.edu/~rucheek

http://multicore-swarm.sourceforge.net

SWARM is a portable open source library of basic primitives that provide framework for designing algorithms on multicore systems. Using this framework, we have implemented efficient parallel algorithms for

- important primitive operations such as prefix-sums, pointer-jumping, symmetry breaking, and list ranking;
- combinatorial problems such as sorting and selection;
- parallel graph theoretic algorithms such as spanning tree, minimum spanning tree, graph decomposition, and tree contraction; and
- computational genomics applications such as maximum parsimony.

This documentation provides descriptions for various variables, functions and macros that can be used as API's for developing parallel codes and is supported by the explanation of an example code.

Index of Contents

1. Introduction ………………………………………………………………... 3

1.1 Motivation for SWARM ……………………………………....... …... 3

1.2 What is SWARM ……………………………………………….. ….. 3

2. SWARM Application Programming Interface Tutorial ……………………... 4

2.1 SWARM Program Structure ………………………………………………. 4

2.2 SWARM Variables and Predefines ……………………………………... 6

2.3 SWARM Functions and Primitives ……………………………………...... 7

2.3.1 Initialization, Execution and Termination Functions ……………... 7

2.3.2 Barrier Functions …………………………………………………. 8

2.3.3 Memory Management Functions ………………………………… 10

2.3.4 Broadcast Functions ………………………………………....... 12

2.3.5 Replicate Functions ……………………………………………….. 17

2.3.6 Reduce Functions …………………………………………......... 19

2.3.7 Scan Functions ……………………………………………….......... 20

2.4 SWARM Macros ……………………………………………….. ….. 21

3. Working with the Example Code ……………………………………………… 23

3.1 Standard Deviation ………………………………………………......... 23

3.2 SWARM for Standard Deviation Calculation …………………………….. 23

3.3 Example Code itself ……………………………………………………… 24

3.4 Step by Step Explanation of the Code ……………………………....... 29

4. Conclusion ……………………………………………………………........... 38

1. Introduction:

1.1 Motivation for SWARM

Since the inception of desktop computer, software performance has improved at an exponential rate, primarily driven by rapid growth in processing power. Performance of an algorithm kept on simply improving by the arrival of new and faster processors. However, we can no longer solely rely on Moore's law for performance improvements. Fundamental physical limitations such as the size of the transistor and power constraints have now necessitated a radical change in commodity microprocessor architecture to multicore designs. Dual and quad-core processors are slowly and steadily finding their way into the desktops and the laptops. Software developers and programmers are now required to exploit this concurency at algorithmic level.

1.2 What is SWARM?

- SWARM (**S**oft**W**are and **A**lgorithms for **R**unning on **M**ulticore) has been introduced as an open source parallel programming framework. It is a library of primitives that fully exploit the multicore processors.
- The SWARM programming framework is a descendant of the symmetric multiprocessor (SMP) node library component of SIMPLE (*Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999).
- SWARM is built on POSIX threads that allow the user to use either the already developed primitives or direct thread primitives.
- SWARM has constructs for parallelization, restricting control of threads, allocation and de-allocation of shared memory, and communication primitives for synchronization, replication and broadcasting.
- The framework has been successfully used to implement efficient parallel versions of primitive algorithms. Viz. List ranking, Prefix sums, Symmetry breaking etc.

In order to use the SWARM library, the programmer needs to make minimal modifications to existing sequential code. After identifying compute-intensive routines in the program, work can be assigned to each core using an efficient multicore algorithm. Independent operations such as those arising in functional parallelism or loop parallelism can be typically threaded. For functional parallelism, this means that each thread acts as a functional process for that task, and for loop parallelism, each thread computes its portion of the computation concurrently. Note that it might be necessary to apply loop transformations to reduce data dependencies between threads. SWARM contains efficient implementations of commonly-used primitives in parallel programming. These computation and communication primitives have been discussed below and followed up with their usage in an example code.

2. SWARM Application Programming Interface Tutorial

2.1 SWARM Program Structure

Before beginning with the SWARM API functions and variables, we first present the structure of a typical SWARM program. This will help in understanding the functions along with their usage described later. The code executes the function 'routine' in parallel, synchronizes at a certain point and later prints the thread id on which it is running for each individual thread. 8 threads/cores have been assumed for all the examples presented in this section.

```
#include <swarm.h>

static void routine (THREADED)
{
      ......
      ......

      SWARM_Barrier_sync(TH);

      ......

      printf("My thread id = %d, Total threads = %d\n", MYTHREAD,
THREADS);

      ......
      ......
}


int main (int argc, char **argv)
{
      SWARM_Init(&argc, &argv);
      /* sequential code */
      ......
      ......

      /* parallelize a routine using SWARM */
      SWARM_Run(routine);
      /* more sequential code */

      ......
      ......
      /*Terminate cleanly */
      SWARM_Finalize();
}
```

Corresponding output (assuming 8 threads

```
My thread id = 2, Total threads = 8
My thread id = 5, Total threads = 8
My thread id = 0, Total threads = 8
My thread id = 1, Total threads = 8
My thread id = 7, Total threads = 8
My thread id = 4, Total threads = 8
My thread id = 3, Total threads = 8
My thread id = 6, Total threads = 8
```

The order in which the threads are printed varies on different executions.

2.2 SWARM Variables and Predefines

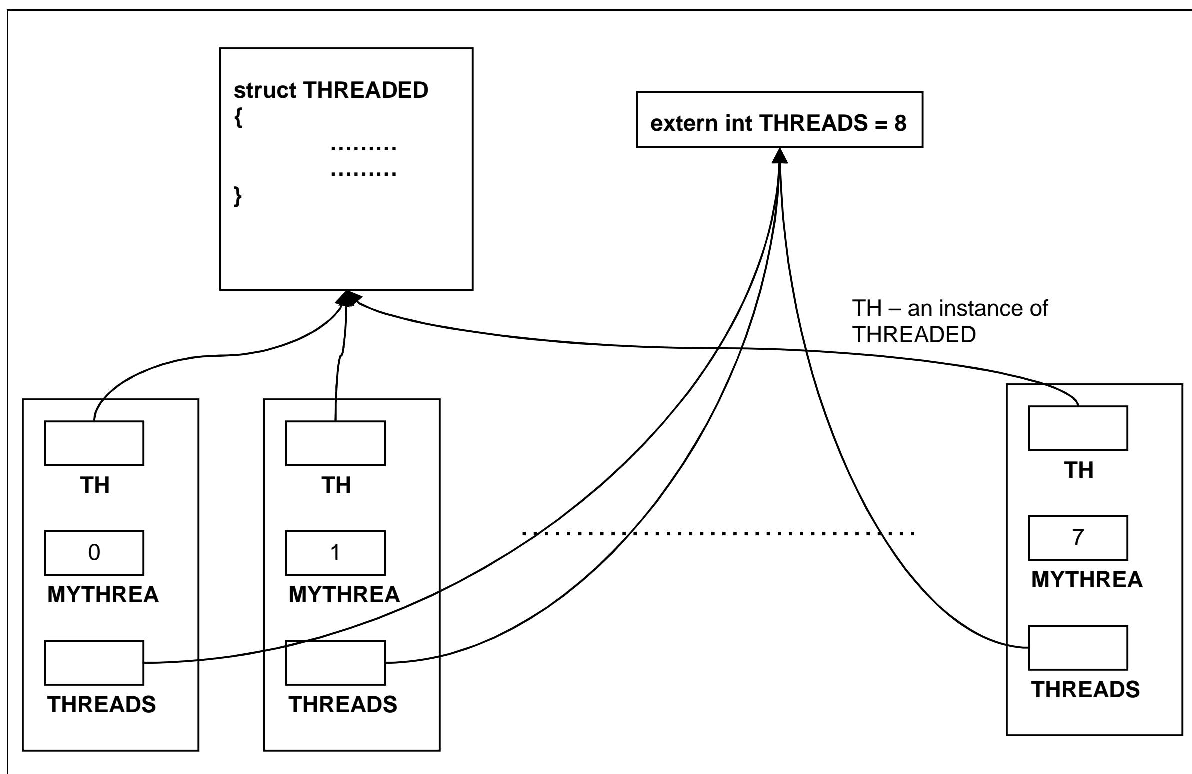
THREADED: THREADED is defined to be a structure that contains all the required information for the particular thread. The parallel routine must be invoked with THREADED as the argument.

TH: TH is instance of THREADED. As seen in the above code, TH is passed as an argument for SWARM_Barrier_sync. The function definition that accepts TH is defined as SWARM_Barrier_sync(THREADED) which can be seen in the swarm.h header file. From the programming point of view, not much has to be worried about TH and THREADED as they have been declared for internal implementation purposes.

MYTHREAD: Provides the thread id of the thread containing it. MYTHREAD is output on the test code for each of the threads.

THREADS: Specifies the total number of threads executing in parallel. Both THREADS and MYTHREAD are useful variables from programmers point of view.

Figure below explains the above concepts:



2.3 SWARM Functions and Primitives

2.3.1 Initialization, Execution and Termination Functions:

These functions are called from the main of the program. As replicated in the code below, most of the SWARM applications will have the structure of main using these three functions.

```
int main (int argc, char **argv)
{
      SWARM_Init(&argc, &argv);
      /* sequential code */
      ......
      ......

      /* parallelize a routine using SWARM */
      SWARM_Run(routine);
      /* more sequential code */

      ......
      ......
      /*Terminate cleanly */
      SWARM_Finalize();
}
```

Each of them is further described below.

- **SWARM_Init(&argc, &argv)**

  This function is responsible for initializing the parallel environment and allocating the requisite memory. It looks at the number of processors available on the machine and sets up the threads accordingly. The user can also specify the number of threads by using the option –t.

- **SWARM_Run ((void *)routine)**

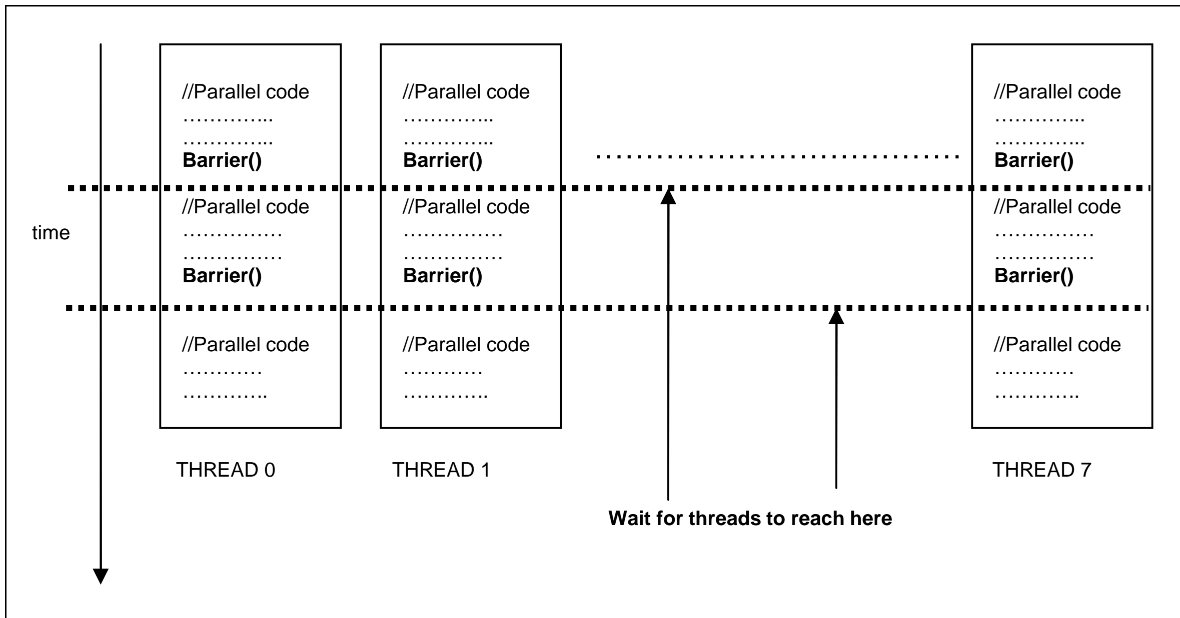
  This is the routine which we want to parallelize across various threads. Thread creation and execution takes place here.

- **SWARM_Finalize()**

  Performs the clean up task by freeing up the allocated memory.

2.3.2 Barrier Functions

Barrier functions are used to synchronize the execution on various threads. The threads execute asynchronously in parallel until this function is called. When one of the threads reaches the barrier, it waits until all the threads reach this point, thus synchronizing all the threads. Once synchronized, all threads start executing asynchronously in parallel again.



- **void SWARM_Barrier()**

  By default SWARM_Barrier uses SWARM_Barrier_sync, which is described next.

Usage:

```
static void routine (THREADED)
{
       /* parallel code */
       ......
       ......

       /* use the SWARM Barrier for synchronization */
       SWARM_Barrier();

       /* more parallel code */
       ......
       ......
}
```

- **void SWARM_Barrier_sync(THREADED)**

This function achieves thread synchronization using conditional wait on mutex lock until all threads have reached.

Usage:

```
static void routine (THREADED)
{
      /* parallel code */
      ......
      ......

      /* use the SWARM Barrier for synchronization */
      SWARM_Barrier_sync();

      /* more parallel code */
      ......
      ......
}
```

- **void SWARM_Barrier_tree(THREADED)**

This function achieves thread synchronization using shared buffers. Threads are viewed as a binary tree based on their thread id. Each thread sends a message to the parent thread when it invokes the barrier. This goes bottom up until root detects completion of barrier. The root, now releases the processes for further execution in top down order.

Usage:

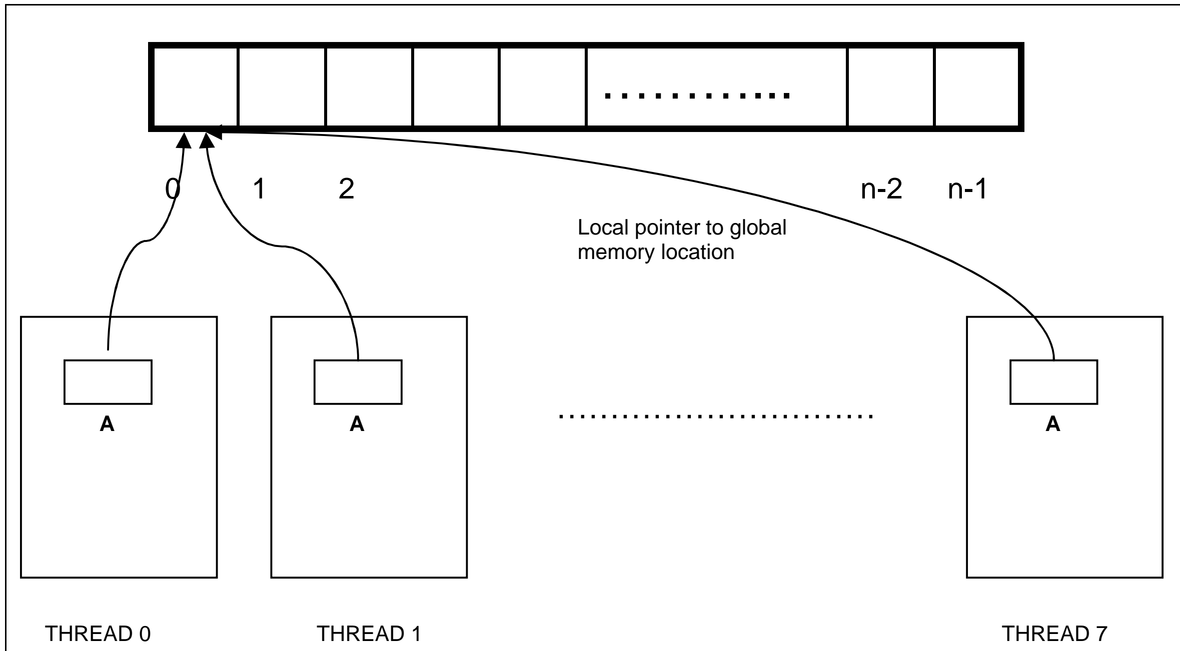
```
static void routine (THREADED)
{
      /* parallel code */
      ......
      ......

      /* use the SWARM Barrier for synchronization */
      SWARM_Barrier_tree();

      /* more parallel code */
      ......
      ......
}
```

2.3.3 Memory Management Functions:

These are wrapper functions for dynamic memory allocation and release of shared structures. Allocation of memory is done globally, i.e. all threads will access the same memory location.



- **void *SWARM_malloc(int bytes, THREADED)**

  Dynamic memory allocation for specified number of bytes. The underlying function implementation ensures that allocation is done only on one of the threads. Return type is a void pointer and can be type-casted to achieve desired type.

Usage:

```
static void routine (THREADED)
{
      int *A;
      ......
      ......

      /* example: allocate a shared array of size n */
      A = (int*)SWARM_malloc(n*sizeof(int),TH);

      ......
      ......
}
```

- **void *SWARM_malloc_l(long bytes, THREADED)**

  Similar to SWARM_malloc, except that, it is useful for allocating larger amounts of memory that cannot be specified by int. Thus, if sizeof(int) is 4 bytes and if we want to allocate more than 2^32 – 1 (or greater than or equal to 2GB of memory), SWARM_malloc_l should be used.

Usage:

```
static void routine (THREADED)
{
      int *A;
      ......
      ......

      /* example: allocate a shared array of size n */
      A = (int*)SWARM_malloc_l(n*sizeof(int),TH);

      ......
      ......
}
```

- **void SWARM_free(void *ptr, THREADED)**

Used for de-allocating the dynamically allocated memory. The underlying function implementation ensures that de-allocation is done only on one of the threads.

Usage:

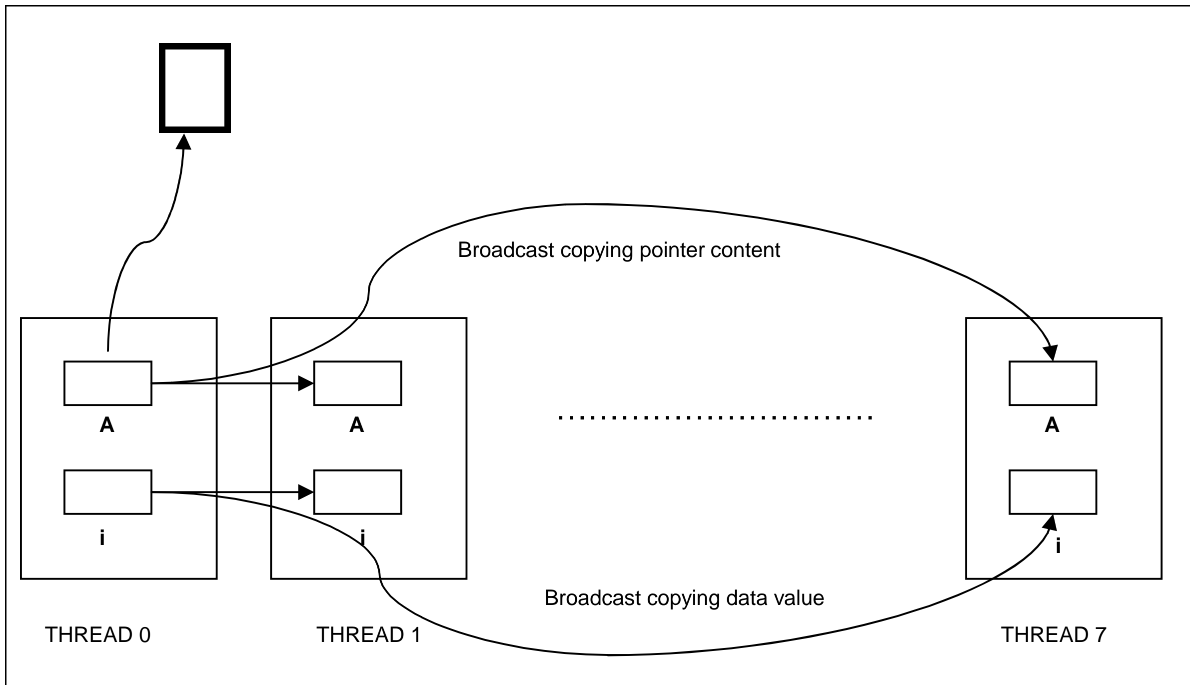
```
static void routine (THREADED)
{
      int *A;
      ......
      ......

      A = (int*)SWARM_malloc(n*sizeof(int),TH);
      ......
      ......

      /* Free the memory allocated for A */
      SWARM_free(A);
      ......
}
```

2.3.4 Broadcast Functions:

There are many situations where a value or a memory location on one of the threads or cores needs to be distributed to other cores. Broadcast functions are used for the same.



- **int SWARM_Bcast_i(int myval, THREADED)**

  Used to Broadcast integer values to all the cores.

Usage:

```
static void routine (THREADED)
{
      int i = 5;
      ......
      ......

      on_one_thread
      {
            i = 10;
            printf("Before broadcast:\n\n");
      }
      printf("Value of i = %d on thread %d\n", I, MYTHREAD);
      SWARM_Barrier();

      /* Broadcasting value of i to all cores */
      i = SWARM_Bcast_i(i,TH);
```

```
        on_one_thread
        {
              printf("\n\n");
              printf("After broadcast:\n\n");
        }
        printf("Value of i = %d on thread %d\n", I, MYTHREAD);
        ......
        ......
}
```

Output for the above code:

```
Before broadcast:

Value of i = 5 on thread 1
Value of i = 10 on thread 0
Value of i = 5 on thread 7
Value of i = 5 on thread 2
Value of i = 5 on thread 3
Value of i = 5 on thread 4
Value of i = 5 on thread 6
Value of i = 5 on thread 5


After broadcast:

Value of i = 10 on thread 1
Value of i = 10 on thread 0
Value of i = 10 on thread 7
Value of i = 10 on thread 2
Value of i = 10 on thread 3
Value of i = 10 on thread 4
Value of i = 10 on thread 6
Value of i = 10 on thread 5
```

- **int SWARM_Bcast_from_i(int myval, int source, THREADED)**

  Broadcast routine mentioned previously, broadcasts the value on thread/core 0. However, in cases where it is required to copy values from a specific thread, you can use the Bcast_from function. Functionally, it is same as Bcast, except that you need to specify the source thread id as the argument.

- **long SWARM_Bcast_l(long myval, THREADED)**

  Broadcast values of type long to all cores.

- **long SWARM_Bcast_from_l(long myval, int source, THREADED)**

  Broadcast values of type long from a specific core to all cores.

- **double SWARM_Bcast_d(double myval, THREADED)**

  Broadcast values of type double to all cores.

- **double SWARM_Bcast_from_d(double myval, int source, THREADED)**

  Broadcast values of type double from a specific core to all cores.

- **char SWARM_Bcast_c(char myval, THREADED)**

  Broadcast char values to all the cores.

- **char SWARM_Bcast_from_c(char myval, int source, THREADED)**

  Broadcast char values from a specific core to all cores.

- **int *SWARM_Bcast_ip(int *myval, THREADED)**

  Used to provide each processing core with the address of the shared buffer. Care should be taken to ensure that the buffer whose address is being broadcasted is shared.

Usage:

```
static void routine (THREADED)
{
    int *a = (int *)SWARM_malloc(sizeof(int), TH);
    int *b = (int *)SWARM_malloc(sizeof(int), TH);
    int *c;

    *a = 5;
    *b = 10;
    ......
    ......

    c = a;


    SWARM_Barrier();
    on_one_thread
    {
         printf("Before broadcast:\n\n");
         c = b;          //b is shared
```

```
    }
    printf("Value of *c = %d on thread %d\n", *c, MYTHREAD);

    /* Broadcasting address of b to all cores */
    c = SWARM_Bcast_ip(c,TH);

    on_one_thread
    {
         printf("\n\n");
         printf("After broadcast:\n\n");
    }
    SWARM_Barrier();
    printf("Value of *c = %d on thread %d\n", *c, MYTHREAD);
}
```

Output for the above code:

```
Before broadcast:

Value of *c = 5 on thread 1
Value of *c = 10 on thread 0
Value of *c = 5 on thread 7
Value of *c = 5 on thread 2
Value of *c = 5 on thread 3
Value of *c = 5 on thread 4
Value of *c = 5 on thread 6
Value of *c = 5 on thread 5


After broadcast:

Value of *c = 10 on thread 1
Value of *c = 10 on thread 0
Value of *c = 10 on thread 7
Value of *c = 10 on thread 2
Value of *c = 10 on thread 3
Value of *c = 10 on thread 4
Value of *c = 10 on thread 6
Value of *c = 10 on thread 5
```

- **int *SWARM_Bcast_from_ip(int *myval, int source, THREADED)**

  Broadcast pointer to an integer from a specific core to all cores.

- **long *SWARM_Bcast_lp(long *myval, THREADED)**

  Broadcast pointer to long integer.

- **long *SWARM_Bcast_from_lp(long *myval, int source, THREADED)**

  Broadcast pointer to a long integer from a specific core to all cores.

- **double *SWARM_Bcast_dp(double *myval, THREADED)**

  Broadcast pointer to a double.

- **double *SWARM_Bcast_from_dp(double *myval, int source, THREADED)**

  Broadcast pointer to a double from a specific core to all cores.

- **char *SWARM_Bcast_cp(char *myval, THREADED)**

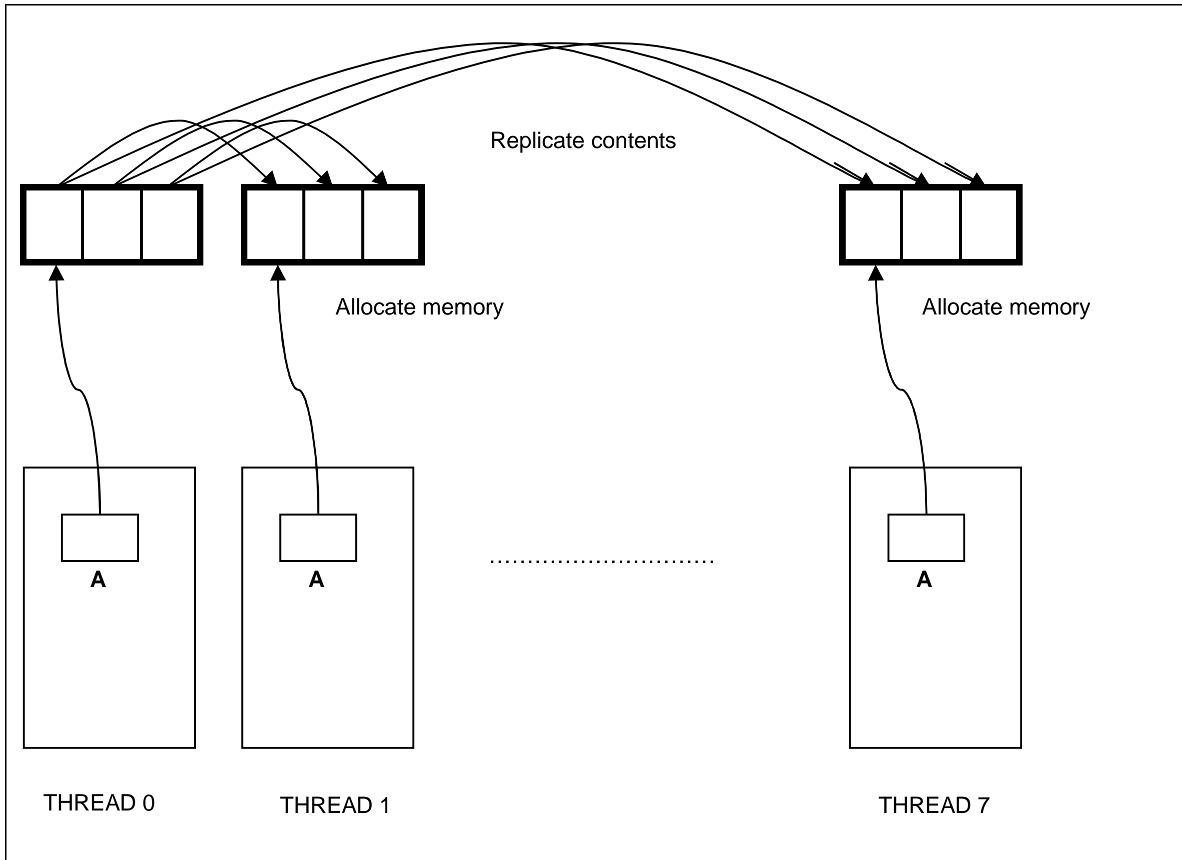
  Broadcast pointer to a character.

- **char *SWARM_Bcast_from_cp(char *myval, int source, THREADED)**

  Broadcast pointer to a character from a specific core to call cores.

2.3.5 Replicate Functions:

The basic difference between replicate and broadcast is that, while broadcast copies the value into pre-existing memory locations on all cores, replicate allocates memory during the function call and creates the replica of the object passed by the calling thread.



- **void *SWARM_Replicate(void *myval, int source, int bytes, THREADED)**

  The arguments for the replicate function require you to pass the pointer to the object to be replicated, the thread/core id of the calling thread and the size of object to be replicated. The returning object must be type-casted with the corresponding type of the passed object.

Usage

```
static void routine (THREADED)
{

    double *a = NULL;
    int size = 0;
```

```
    on_thread(1)
    {
        a = (double *)malloc(3 * sizeof(double));
        a[0] = 9.99;
        a[1] = 8.88;
        a[2] = 7.77;
        size = sizeof(double)*3;
        printf("Before Replicate\n\n");
        printf("On thread %d  values = %lf %lf %lf\n", MYTHREAD, *a,
*(a+1), *(a+2));
    }

    /*Replicating the double object 'a' to on all cores*/
    a = (double *)SWARM_Replicate(a, 1, size, TH);

    SWARM_Barrier();
    on_one_thread
    {
        printf("\n\n");
        printf("After Replicate\n\n");
    }

    SWARM_Barrier();

    printf("On thread %d  values = %lf %lf %lf\n", MYTHREAD, *a,
*(a+1), *(a+2));
}
```

Output:

```
Before Replicate

On thread 1  values = 9.990000 8.880000 7.770000


After Replicate

On thread 0  values = 9.990000 8.880000 7.770000
On thread 1  values = 9.990000 8.880000 7.770000
On thread 2  values = 9.990000 8.880000 7.770000
On thread 3  values = 9.990000 8.880000 7.770000
On thread 4  values = 9.990000 8.880000 7.770000
On thread 5  values = 9.990000 8.880000 7.770000
```

2.3.5 Reduce Functions:

These are set of functions that are used to obtain addition, maximum or minimum of values calculated across different threads. Each thread provides the value in its local copy to this primitive and the operation specified by the 'op' argument is performed on these values. op can take values SUM, MAX or MIN based on task to be performed. Used for integer valued operations.

- **int SWARM_Reduce_i(int myval, reduce_t op, THREADED)**

  The value provided by each thread is of type integer.

Usage:

```
static void routine (THREADED)
{
      int sum = 0;

      ......
      ......

      sum = SWARM_Reduce_i(MYTHREAD, SUM, TH);
      printf("Value of sum = %d on thread %d\n", sum, MYTHREAD);

      ......
}
```

Output:

```
Value of sum = 28 on thread 0
Value of sum = 28 on thread 1
Value of sum = 28 on thread 2
Value of sum = 28 on thread 3
Value of sum = 28 on thread 4
Value of sum = 28 on thread 5
Value of sum = 28 on thread 6
Value of sum = 28 on thread 7
```

- **long SWARM_Reduce_l(long myval, reduce_t op, THREADED)**

  Perform operation on long values.

- **double SWARM_Reduce_d(double myval, reduce_t op, THREADED)**

  Perform operation on floating point values.

2.3.6. Scan Functions:

Scan functions perform task similar to Reduce functions. However, these are prefix operations, in the sense that output of each thread is based only on the threads before them (i.e. those threads having smaller thread id than it).

- **int SWARM_Scan_i(int myval, reduce_t op, THREADED)**

  The value provided by each thread is of type integer.

Usage:

```
static void routine (THREADED)
{
       int sum = 0;

       ......
       ......

       sum = SWARM_Scan_i(MYTHREAD, SUM, TH);
       printf("Value of sum = %d on thread %d\n", sum, MYTHREAD);

       ......
       ......
}
```

Output:

```
Value of sum = 0 on thread 0
Value of sum = 1 on thread 1
Value of sum = 2 on thread 2
Value of sum = 6 on thread 3
Value of sum = 10 on thread 4
Value of sum = 15 on thread 5
Value of sum = 21 on thread 6
Value of sum = 28 on thread 7
```

- **long SWARM_Scan_l(long myval, reduce_t op, THREADED)**

  Perform operation on long values.

- **double SWARM_Scan_d(double myval, reduce_t op, THREADED)**

  Perform operation on float values.

2.4 Macros for SWARM

- **on_thread, on_one_thread**

Control can be given to any particular thread using these macros. on_one_thread gives control to thread 0.

Usage:

```
static void routine (THREADED)
{
       ......
       ......

       /* example: execute code on thread MYTHREAD */
       on_thread(THREADS - 1)
       {
             printf("Reached here in only one of the threads\n");
             printf("In thread %d\n\n", MYTHREAD);
       }
       SWARM_Barrier();

       /* example: execute code on one thread */
       on_one_thread
       {
             printf("Reached here in only one of the threads\n");
             printf("In thread %d\n", MYTHREAD);
       }

       ......
       ......
}
```

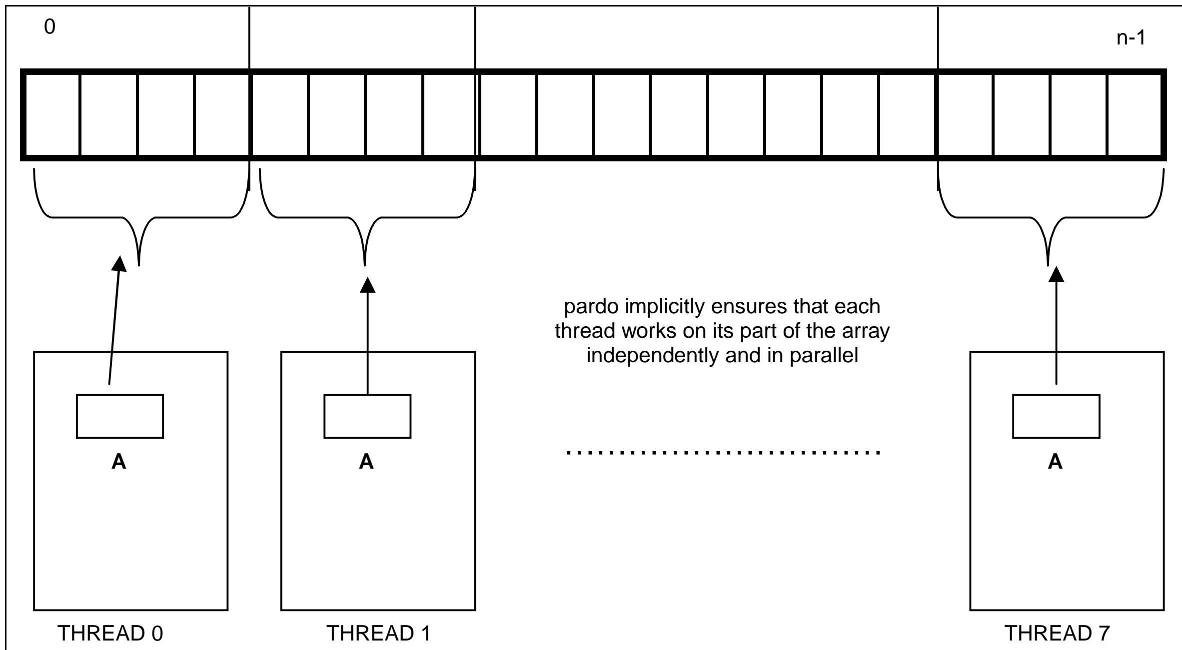
Output:

```
Reached here in only one of the threads
In thread 7

Reached here in only one of the threads
In thread 0
```

- **SWARM_pardo**

  The SWARM library contains several basic "pardo" directives for executing loops concurrently on one or more processing cores. Typically, this is useful when an independent operation is to be applied to every location in an array, for example element-wise addition of two arrays. Pardo implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores.



Usage:

```
static void routine (THREADED)
{
        ......
        /*example: partitioning a "for" loop among cores */
        SWARM_pardo(i, start, end, incr)
        {
                A[i] = A[i] * A[i];
        }
        ......
}
```

3. Working with the Example Code

The example code presented below is used to demonstrate the SWARM API. At least one function of each type (synchronization, replication, broadcast etc.) has been incorporated into the example code.

The code is used to calculate the standard deviation of a set of numbers represented in an array in a parallel environment. The standard deviation is used to measure how widely the data is spread in a distribution.

3.1 Standard Deviation (σ)

If $x_1, x_2, x_3 \ldots x_n$ represents a sequence of 'n' numbers, then standard distribution 'σ' is given mathematically as:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}}$$

where μ is the mean of the distribution given by,

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

3.2 SWARM for Standard Deviation Calculation

The basic idea is to distribute the elements in the array across various processors/threads. Each thread calculates the sum on its part of the array, which is used to calculate the total sum and hence, the mean. The mean is now distributed across various threads, after which each thread now computes the sum of square of differences with the mean on its part of the array, which is used to calculate the final standard deviation.

The example code has been replicated below followed by which, we have a detailed explanation of the code.

3.3 The Code itself

```
#include <swarm.h>
#include <swarm_random.h>

static void stddev_routine (THREADED)
{
	int i, n = 10;
	int max_random = 10, partial_sum = 0, prefix_sum = 0, global_sum
= 0;
	int *arr;
	double global_mean = 0, partial_squared_sum = 0, squared_sum = 0,
std_dev = 0;

	arr = SWARM_malloc(n * sizeof(int), TH);
	on_one_thread
	{
		printf("Array memory allocated on a single thread...\n\n");
	}

	/***************** Figure 3.4.1 ************************/

	SWARM_random_init(TH);
	SWARM_srandom(MYTHREAD + 1,TH);

	pardo(i, 0, n, 1)
	{
		arr[i] = SWARM_random(TH)%max_random;
	}

	SWARM_Barrier();

	/***************** Figure 3.4.2 ************************/


	on_one_thread
	{
		printf("Randomly generated array is:\n");
		for(i = 0; i < n; i++)
		{
			printf("arr[%d] = %ld\n", i, arr[i]);
		}
		printf("\n\n");
	}

	SWARM_Barrier();

	pardo(i, 0, n, 1)
	{
		partial_sum += arr[i];
	}
```

```
	on_one_thread
	{
		printf("Partial Sum:\n");
	}

	printf("Thread %d: %d\n", MYTHREAD, partial_sum);

	/***************** Figure 3.4.3 ***********************/


	SWARM_Barrier();

	on_one_thread
	{
		printf("\n\n");
		printf("Global sum calculated using Reduce
operation...\n\n");
		printf("Global Sum:\n");
	}

	SWARM_Barrier();
	global_sum = SWARM_Reduce_i(partial_sum, SUM, TH);
	printf("Thread %d: %d\n", MYTHREAD, global_sum);

	/***************** Figure 3.4.4 ***********************/

	on_one_thread
	{
		global_mean = 1.0 * global_sum/n;
		printf("\n\n");
		printf("Mean calculated on a thread = %f\n", global_mean);
		printf("\n\n");
		printf("Broadcasting mean...\n\n");
		printf("Global Mean:\n");
	}

	/***************** Figure 3.4.5 ***********************/


	global_mean = SWARM_Bcast_d(global_mean, TH);

	/***************** Figure 3.4.6 ***********************/


	SWARM_Barrier();
	printf("Thread %d: %f\n", MYTHREAD, global_mean);
	SWARM_Barrier();

	pardo(i, 0, n, 1)
	{
		partial_squared_sum += (arr[i] - global_mean)*(arr[i] -
global_mean);
	}

	/***************** Figure 3.4.7 ***********************/
```

```
	SWARM_Barrier();

	on_one_thread
	{
		printf("\n\n");
		printf("Partial Squared Sum:\n");
	}
	SWARM_Barrier();
	printf("Thread %d: %f\n", MYTHREAD, partial_squared_sum);
	SWARM_Barrier();

	on_one_thread
	{
		printf("\n\nCalculating total squared sum from partial
squared sums using Scan operation...\n\n");
		printf("Total Squared Sum:\n");
	}

	SWARM_Barrier();
	squared_sum = SWARM_Scan_d(partial_squared_sum, SUM, TH);

	printf("Thread %d: %f\n", MYTHREAD, squared_sum);

	/***************** Figure 3.4.8 ***********************/


	SWARM_Barrier();

	on_thread(THREADS - 1)
	{
		printf("\n\n");
		printf("Calculating Standard Deviation on last
thread...\n\n");
		std_dev = sqrt(squared_sum / n);
		printf("Standard Deviation = %f\n", std_dev);
	}

	SWARM_Barrier();

	/***************** Figure 3.4.9 ***********************/


	on_one_thread
	{
		printf("\n\n");
		printf("Releasing array memory...\n\n");
	}
	SWARM_free(arr, TH);
}
```

```c
int main (int argc, char **argv)
{
	SWARM_Init(&argc,&argv);

	SWARM_Run ((void *)stddev_routine);

	SWARM_Finalize();

	return 0;
}
```

Output:

```
THREADS: 3

Array memory allocated on a single thread...

Randomly generated array is:
arr[0] = 3
arr[1] = 4
arr[2] = 7
arr[3] = 8
arr[4] = 5
arr[5] = 9
arr[6] = 0
arr[7] = 6
arr[8] = 2
arr[9] = 2


Partial Sum:
Thread 1: 22
Thread 2: 10
Thread 0: 14


Global sum calculated using Reduce operation...

Global Sum:
Thread 2: 46
Thread 0: 46
Thread 1: 46


Mean calculated on a thread = 4.600000


Broadcasting mean...

Global Mean:
Thread 2: 4.600000
Thread 1: 4.600000
Thread 0: 4.600000
```

```
Partial Squared Sum:
Thread 0: 8.680000
Thread 1: 31.080000
Thread 2: 36.640000


Calculating total squared sum from partial squared sums using Scan
operation...

Total Squared Sum:
Thread 1: 39.760000
Thread 0: 8.680000
Thread 2: 76.400000


Calculating Standard Deviation on last thread...

Standard Deviation = 2.764055


Releasing array memory...
```

3.4 Step by Step Explanation of the Code

The code execution begins with main. The main construct will be almost similar for all the examples using SWARM. It begins with the initialization of the SWARM parallel environment and concludes with the clean up of this environment. Between this, we have a call to a routine that needs to be executed in this parallel environment. All this is achieved using 3 functions – SWARM_Init, SWARM_Run and SWARM_Finalize which are further detailed in the API explanation.

The routine defined for parallelizing the calculation of standard deviation is stddev_routine. This routine will be executed on each of the threads/cores. We now describe an instance of execution. For the purpose of explanation, we assume that there are 3 threads and the size of the data set is 10. (THREADS = 3, n = 10). Within the code, comments have been placed specifying the figure to refer, which will provide a detailed view of code execution at that point of time.

The memory that will hold the data set of numbers is dynamically allocated. This ensures, that only one copy of this list is maintained across all the threads. On the other hand, variables defined locally (viz. n, partial_sum etc) are replicated within all the threads. Figure below supports the explanation.

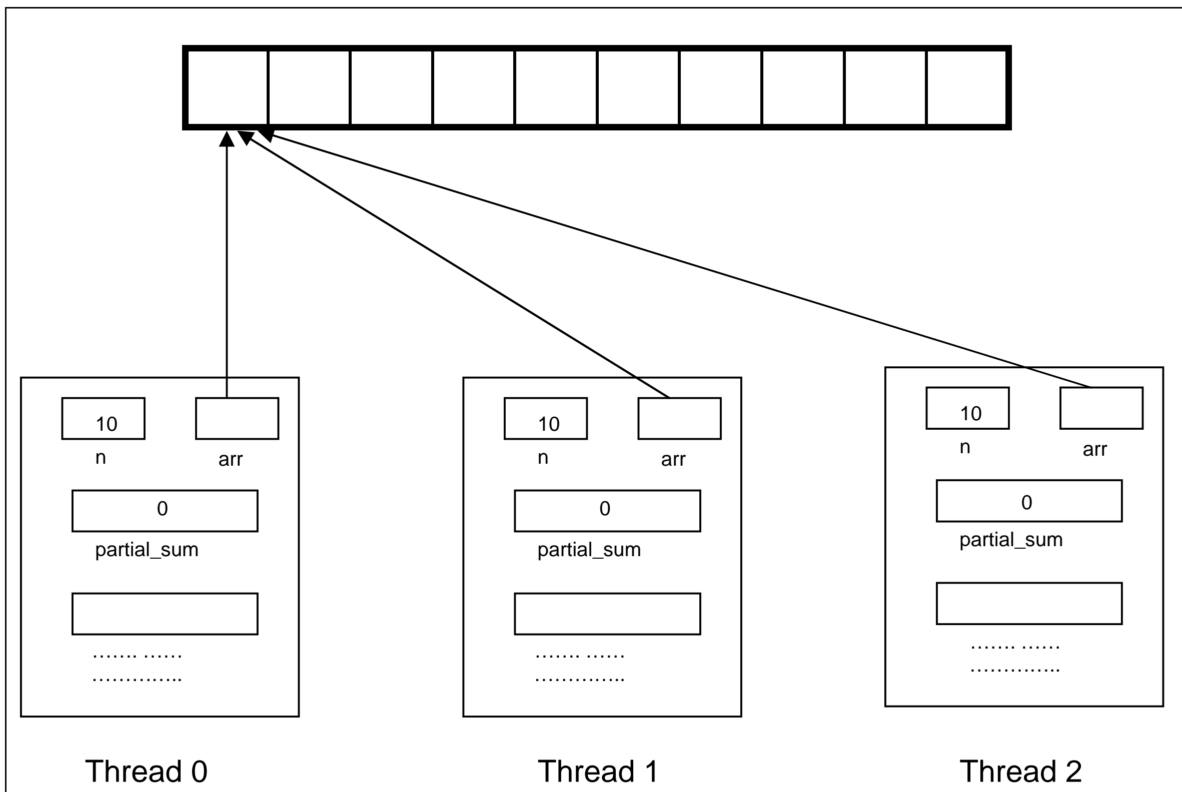


Figure 3.4.1

Next, the array is populated randomly. Randomization functions have also been incorporated in SWARM which mimics the standard random function. A seed has to be specified for generating the random sequence. Varying the seed varies the sequence created. The populating process is also done in a parallel manner. Thus thread 0 fills the first 3 elements, thread 1 fills the next 3 and the last thread fills the remaining elements.

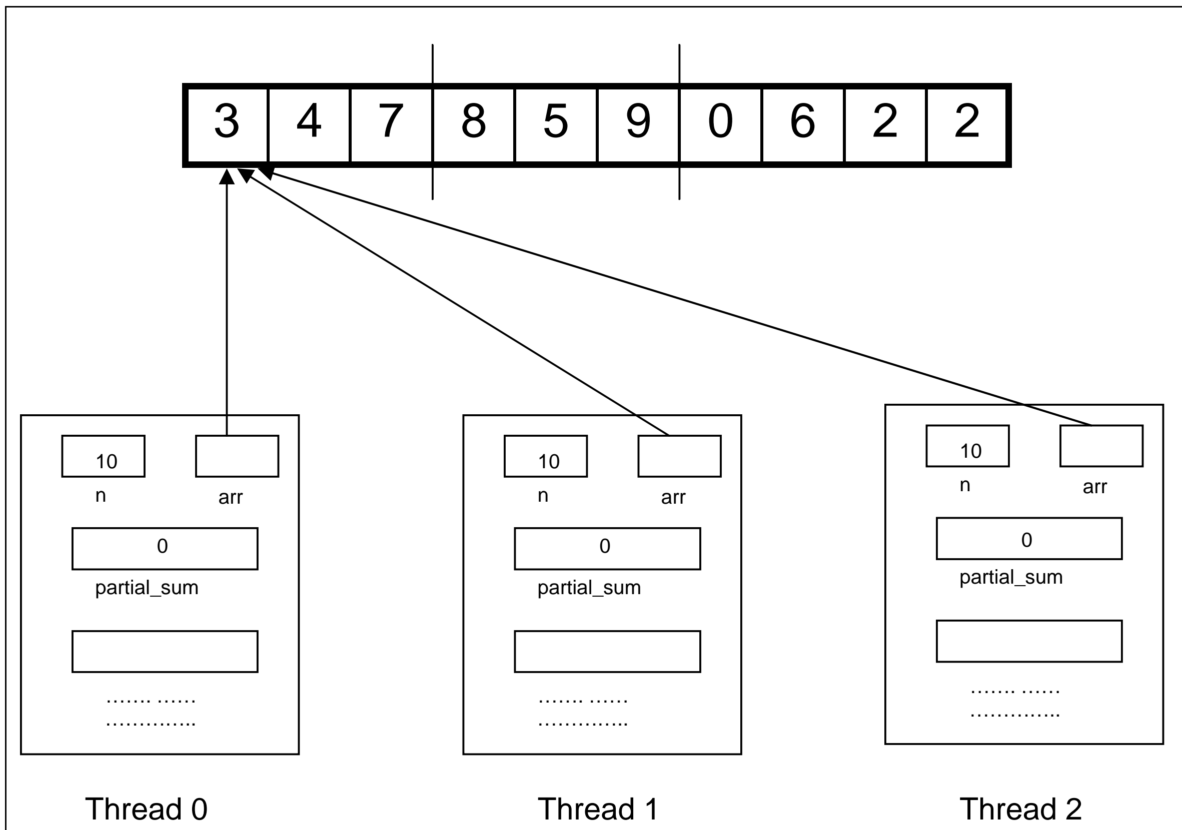


Figure 3.4.2

Each thread now works on its part of the array to calculate the partial sum.

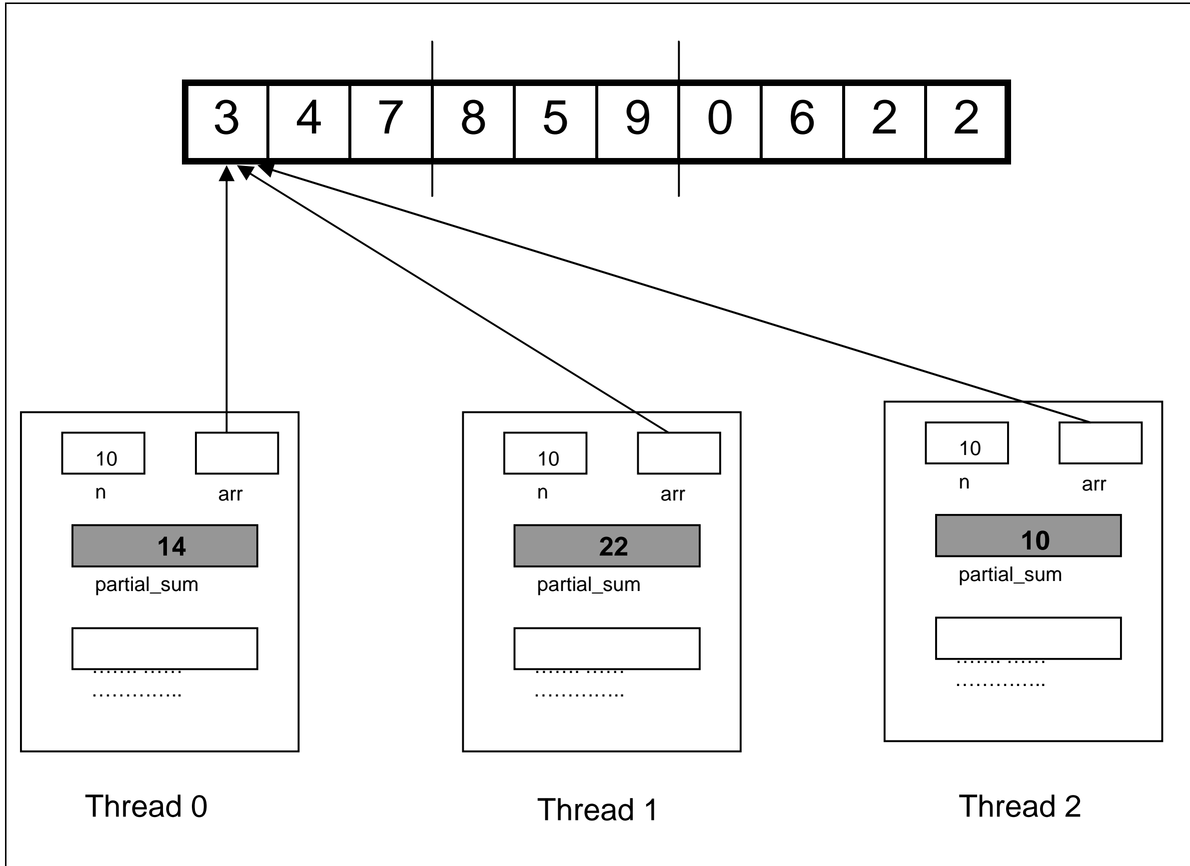


Figure 3.4.3

Once the partial sum is calculated at each thread, global sum is calculated using the reduce operation. Reduce function picks up a value from each thread and performs a binary operation on those values. Here we request to perform sum operation on the partial sum values to obtain the global sum.

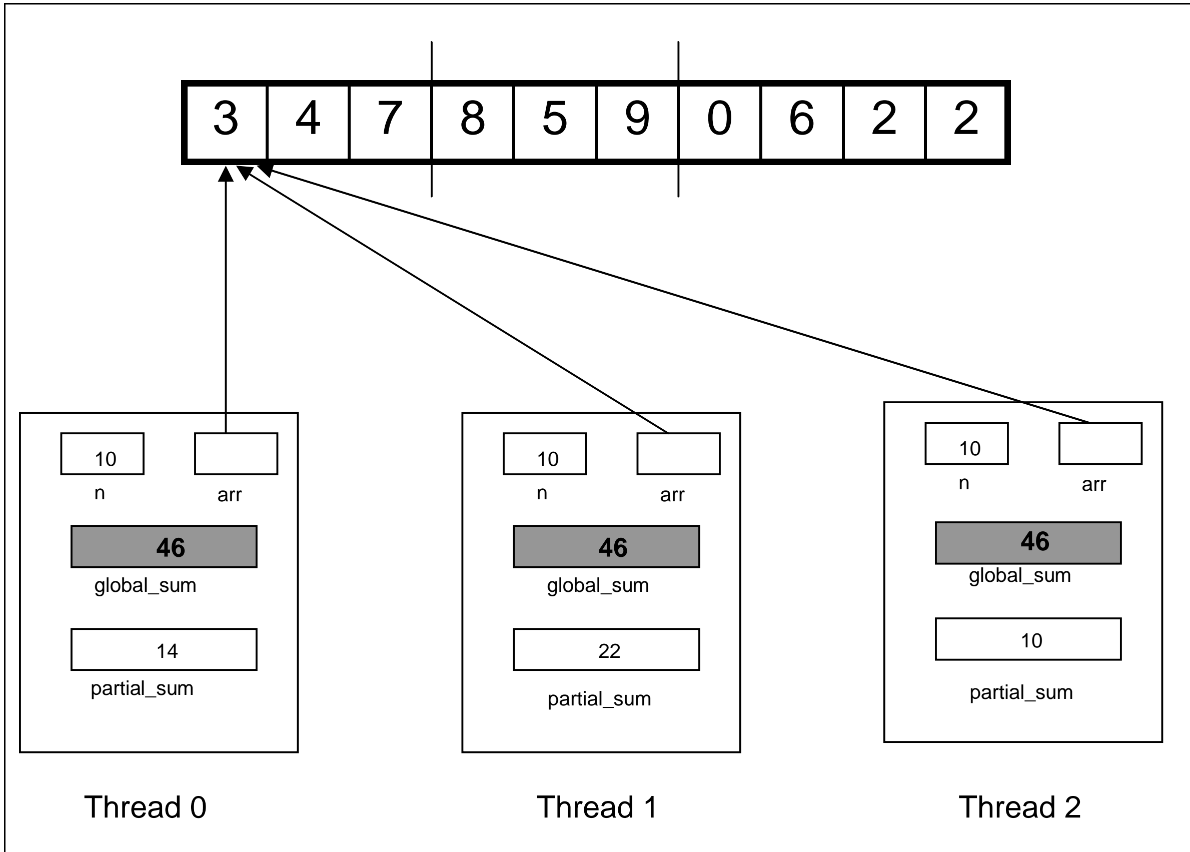


Figure 3.4.4

Global average or global mean is now calculated on thread 0.

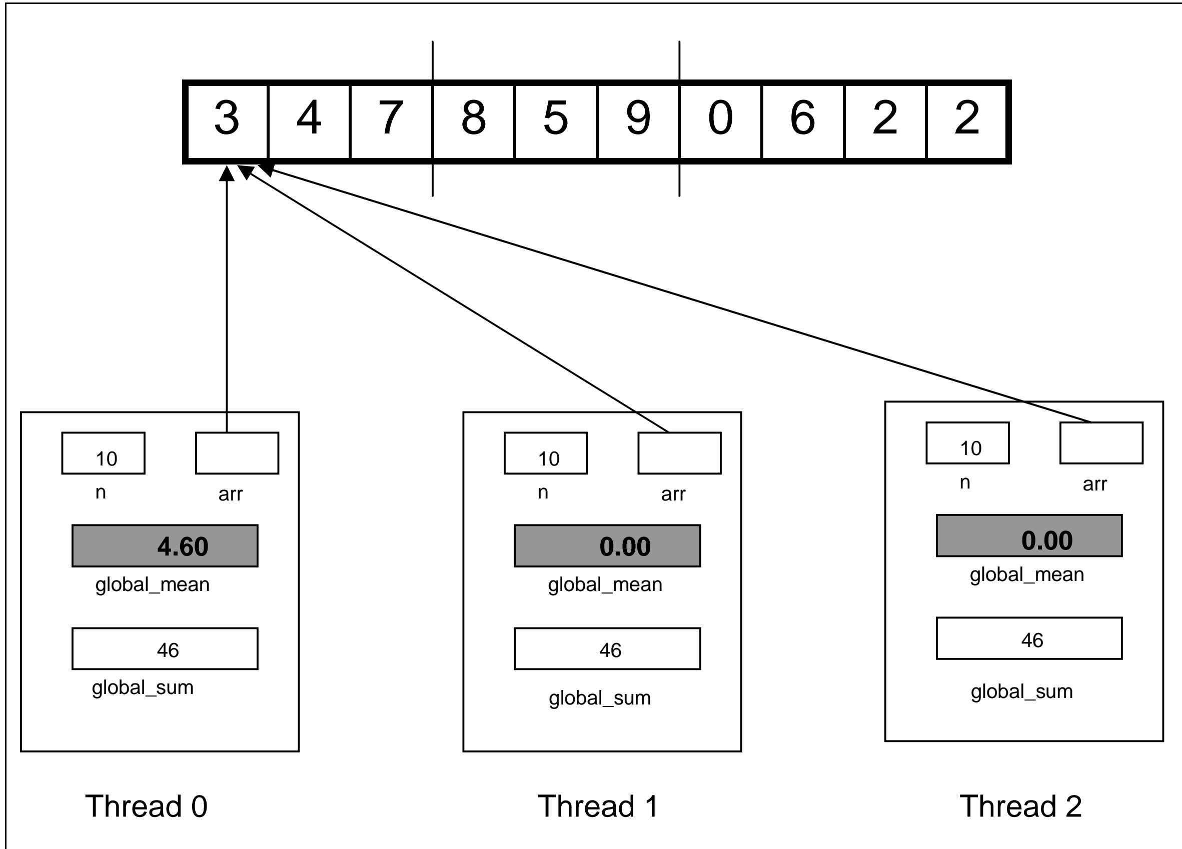


Figure 3.4.5

This global mean on thread 0 has to be broadcasted to all other threads for further calculation of standard deviation.

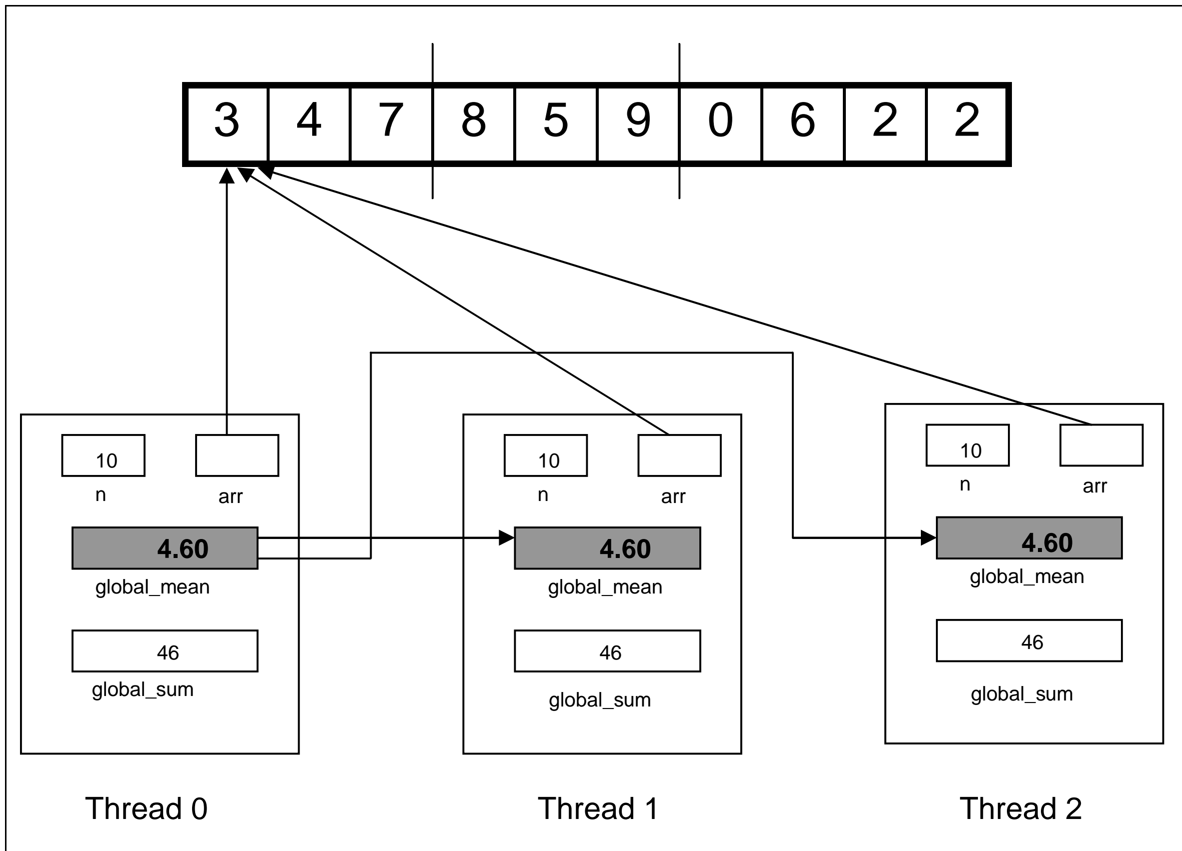


Figure 3.4.6

As each thread receives the value of mean, they now start calculating the sum of square of differences with the mean, on their part of the array. This step is exactly similar to the part where we were calculating the partial sum on each thread.

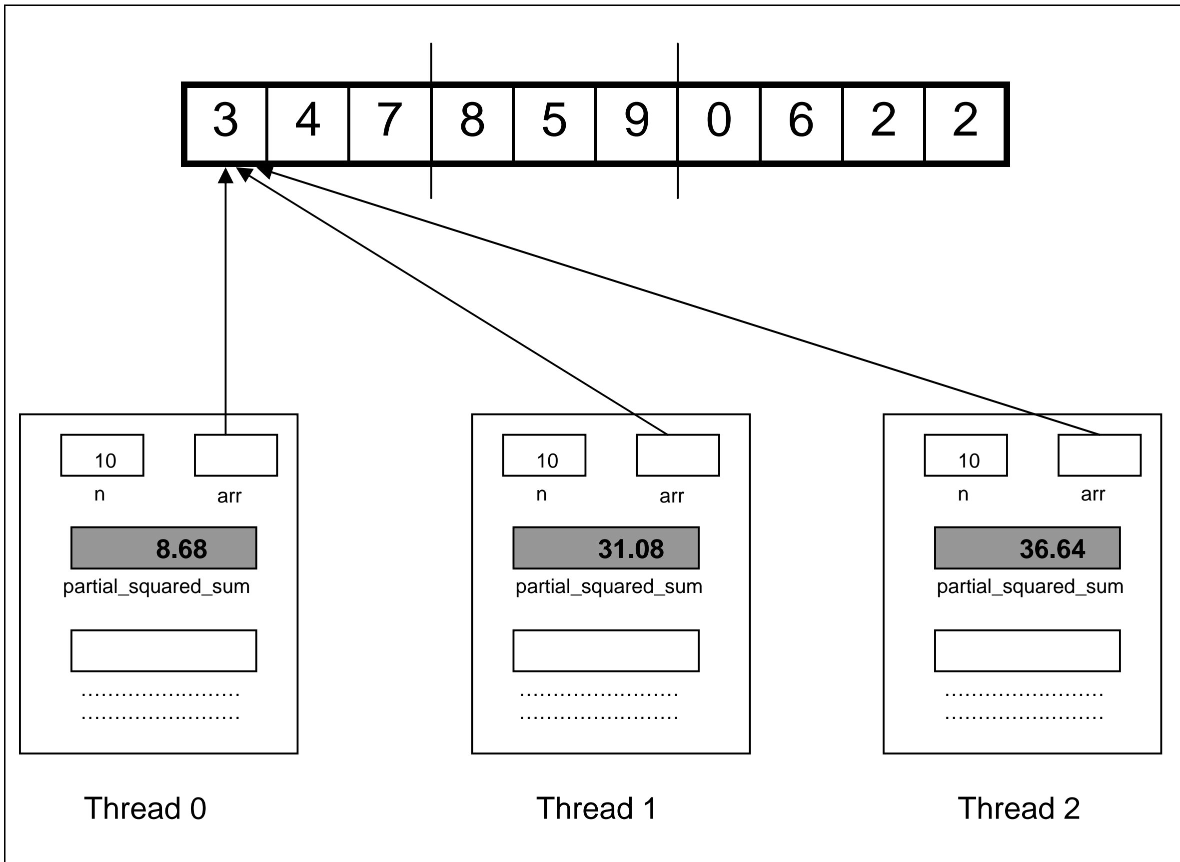


Figure 3.4.7

Scan operation is now used to add up the partial sums. The basic difference between scan and reduce operation is the fact that scan performs prefix operation. Thus thread 0 has its own value, thread 1 has sum of thread 0 and thread 1, while thread 2 has sum of thread 0, 1 and 2. Figure below describes this.

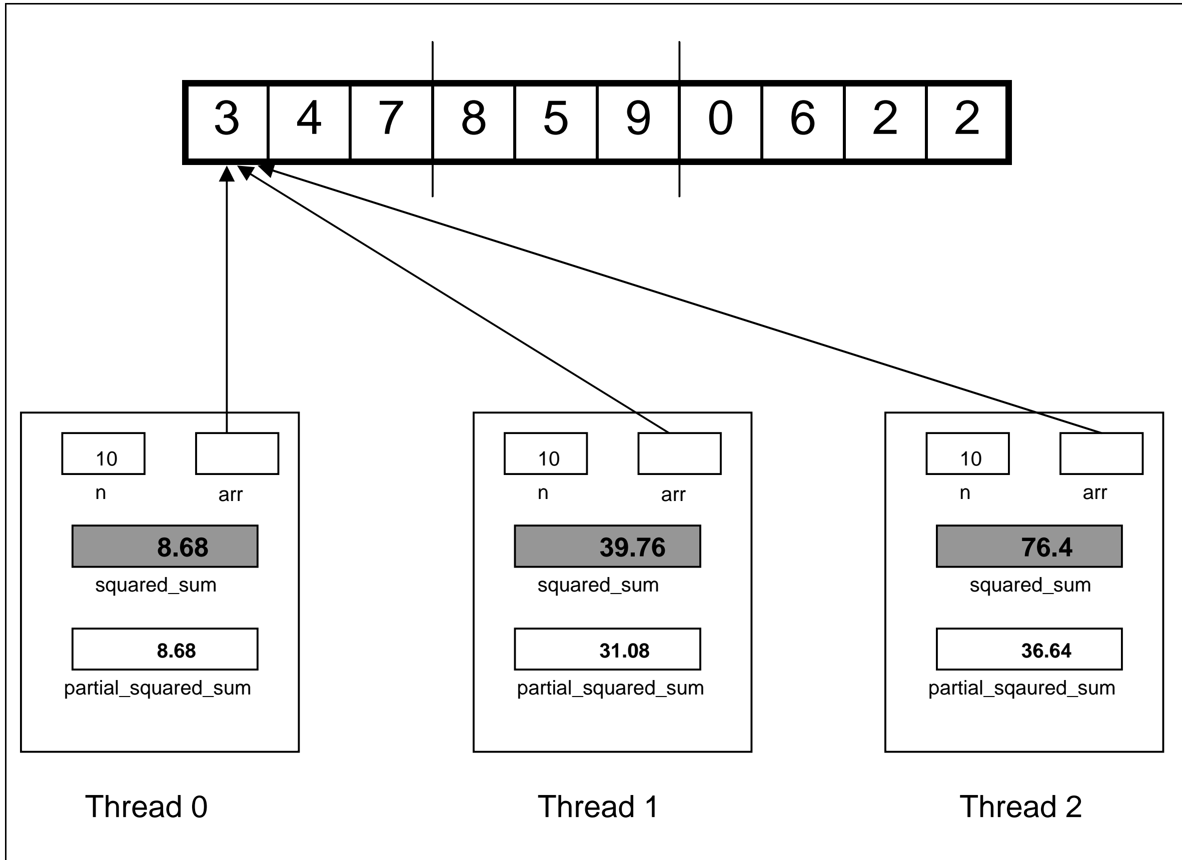


Figure 3.4.8

Finally, the last thread (thread 2 here) has the total squared sum of differences with the mean. We calculate the standard deviation on this thread by dividing the total by the number of elements (10) and then taking the square root.

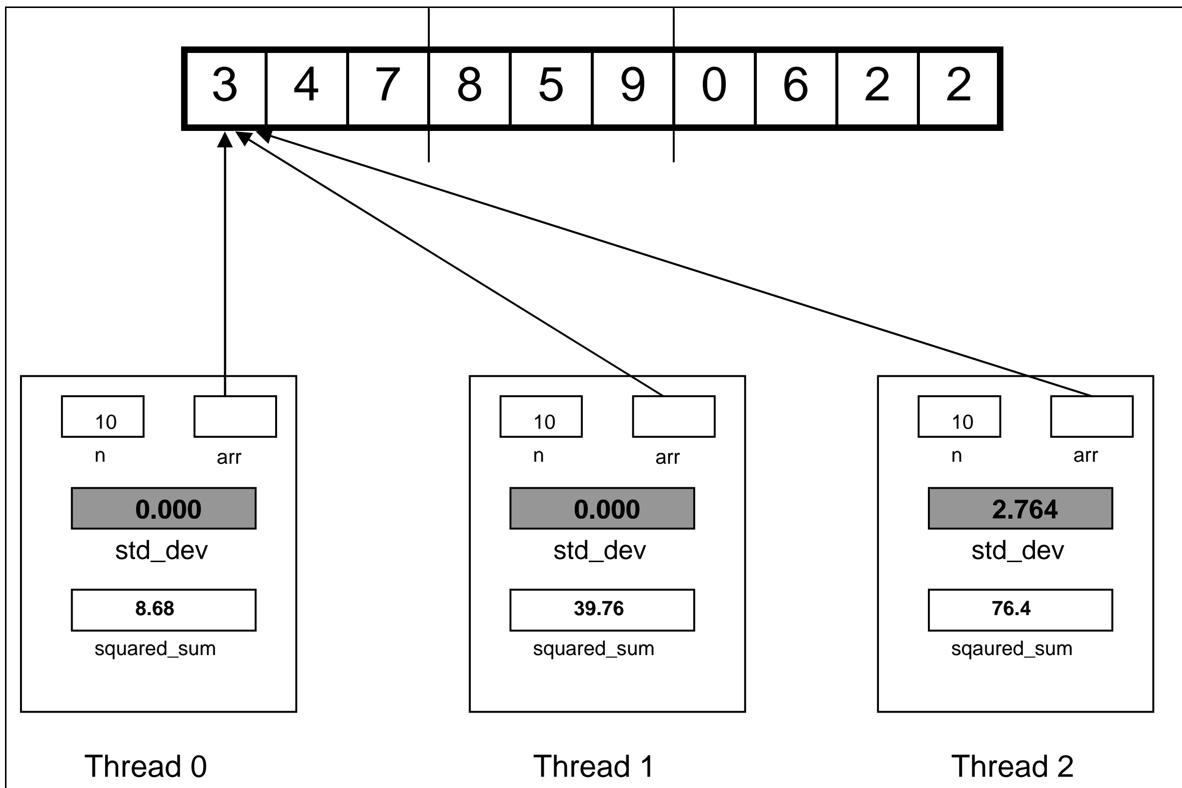


Figure 3.4.9

Once the standard deviation is calculated, the memory allocated for the array is released back to the operating system and we exit the parallel routine.

4. Conclusion

SWARM is thus able to provide a sound interface for program developers to develop parallel applications without worrying about the underlying thread level aspects.
We have already implemented several important parallel primitives and algorithms using this framework. In future, we intend to add to the functionality of basic primitives in SWARM, as well as build more multicore applications using this library.

For further details, you can view the paper: *SWARM: A Parallel Programming Framework for Multicore Processors, David A. Bader, Varun Kanade and Kamesh Madduri*