

MPAC BENCHMARKING LIBRARY

Table of Contents

Introduction	3
1 mpac.h	4
1.1 MPAC_SUCCESS	4
1.2 MPAC_FAILURE	4
1.3 MPAC_NULL	4
1.4 mpac_min	4
1.5 mpac_max	4
1.6 mpac_G	5
1.7 mpac_M	5
1.8 mpac_K	5
1.9 mpac_handle	6
1.10 mpac_init	6
2 mpac_io_file.h	7
2.1 mpac_io_file_length	7
2.2 mpac_io_file_read	7
2.3 mpac_io_file_readln	7
3 mpac_io_net.h	8
3.1 mpac_io_net_handler	8
3.2 mpac_io_net_init	10
3.3 mpac_io_net_getipaddr	10
3.4 mpac_io_net_getportno	10
3.5 mpac_io_net_getproto	11
3.6 mpac_io_net_getdn	11
4 mpac_io_tcp.h	12
4.1 mpac_io_tcp_open_sender	12
4.2 mpac_io_tcp_send	12
4.3 mpac_io_tcp_open_receiver	13
4.4 mpac_io_tcp_accept_receiver	13
4.5 mpac_io_tcp_receive	13
4.6 mpac_io_tcp_close	14
5 mpac_io_udp.h	15
5.1 mpac_io_udp_open_sender	15
5.2 mpac_io_udp_send	15
5.3 mpac_io_udp_open_receiver	15
5.4 mpac_io_udp_receive	16
5.5 mpac_io_udp_close	16
6 mpac_resolution.h	17
6.1 mpac_resval	17
6.2 mpac_resolution_gtd	17
6.3 mpac_resolution_rtc	18
6.4 mpac_resolution_rtc_com	18

7	mpac_stat.h	19
	7.1 mpac_stats_mode	19
	7.2 mpac_stats_mean	19
	7.3 mpac_stats_median	19
	7.4 mpac_stats_min	20
	7.5 mpac_stats_max	20
	7.6 mpac_stats_var	20
	7.7 mpac_stats_std	20
	7.8 mpac_stats_conf_intr	21
	7.9 mpac_stats_cmp	21
8	mpac_thread_manager.h	22
	8.1 mpac_thread_manager_t	22
	8.2 mpac_lock_t	23
	8.3 mpac_thread_manager_init	24
	8.4 mpac_thread_manager_fork	24
	8.5 mpac_thread_manager_wait	24
	8.6 mpac_thread_manager_startj	25
	8.7 mpac_thread_manager_startd	25
	8.8 mpac_thread_manager_end	25
	8.9 mpac_thread_manager_isolate	26
	8.10 mpac_thread_manager_sendsig	26
	8.11 mpac_thread_manager_initlock	26
	8.12 mpac_thread_manager_destroylock	26
	8.13 mpac_thread_manager_barrier	27
	8.14 mpac_thread_manager_No_of_cores	27
	8.15 mpac_thread_manager_core_affnity	27
9	mpac_time.h	28
	9.1 mpac_time_wctime	28
	9.2 mpac_time_fit_gtd	28
	9.3 mpac_time_fit_rtc	29
	9.4 mpac_time_fcpu	29
	9.5 mpac_time_ggettime	29
	9.6 mpac_time_rgettime	30

Introduction

Multicore Processor Architecture and Communication (MPAC) library is specifically designed and developed to ease the development of micro-benchmarks for state-of-the-art commercial multicore systems. MPAC library contains general purpose benchmarking modules that could be used to implement a specialized micro-benchmark. It, not only, encompasses performance measurement methodologies needed for the evaluation of Multicore Systems but, at the same time, is modular enough to be used for any other parallel application.

MPAC uses multithreaded approach so that, if necessary, all the cores of a Multicore system could be exercised simultaneously. MPAC user has full freedom to choose the size of thread pool. It is an open source library that is freely available under FreeBSD style licensing model. It is being developed in C on UNIX based systems using GCC compiler collections but MPAC library is equally beneficial for benchmarking traditional single-core as well as more recent multi-core processors architecture and high performance networking architecture to test their high performance capabilities. In addition, it is equally useful for uniprocessor as well as multiprocessor (SMP) systems. Unlike existing micro-benchmarks and application benchmarks that hardly combine all these areas, which are becoming important due to technology trends in processor and network architectures, MPAC will enable the future benchmarks to provide a holistic view of system performance. Benchmarks developed using MPAC will be equally useful for system architects as well as end users, to compare various systems on the basis of their performance potential.

Hence MPAC is designed to implement performance tool for any general purpose multi-core architecture that evaluates the performance of cache and memory subsystem, CPU and high performance networks, to identify and minimize performance gaps and bottlenecks to attain sustainable maximum performance of multi-core systems and high performance networks collectively.

MPAC library is multithreaded, modular, flexible and extensible, developed in C, to accommodate future parallel benchmarks and applications.

1 mpac.h

Initialization of common MPAC Tools related servies.

Author: Abdul Waheed.

1.1 MPAC_SUCCESS

```
#define MPAC_SUCCESS 0
```

1.2 MPAC_FAILURE

```
#define MPAC_FAILURE -1
```

1.3 MPAC_NULL

```
#define MPAC_FAILURE -1
```

1.4 mpac_min

```
#define mpac_min (a,b)
```

Returns smaller value.

1.5 mpac_max

```
#define mpac_max (a,b)
```

Returns larger value.

1.6 mpac_G

```
#define mpac_G
```

```
#define mpac_G 1000000000.0F
```

1.7 mpac_M

```
#define mpac_M
```

```
#define mpac_M 1000000.0F
```

1.8 mpac_K

```
#define mpac_K
```

```
#define mpac_K 1000.0
```

1.9 struct mpac_handle

Members

- int fd_err
- int fd_limit
- int num_processor

int fd_err

int fd_limit

int num_processor

Number of processors.

1.10 int mpac_init (struct mpac_handle* handle)

Any MPAC Tools related services initial setup is handled by this function.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* MPAC network handler.

2 mpac_io_file.h

File I/O related services.

This file contains the prototype of the functions which can be used to find out the size of a file in bytes, to read a particular line from a file into a buffer and to read a whole file in a buffer.

Author: Abdul Waheed

2.1 int mpac_io_file_length (char* p_file_name)

This function returns the size of a file in bytes. If the file name contains null or the function fails to find out the size of the file then it returns MPAC_FAILURE.

Return Value: Size of a file in bytes or MPAC_FAILURE.

Parameters: *p_file_name* Name of the file.

2.2 int mpac_io_file_read (char* p_file_name, unsigned char* buffer, int buff_size)

This function takes a file name, a pointer to a byte buffer, and length of the file in bytes as input arguments. It then opens and reads the entire file, such that the number of bytes read and copied in the buffer is equal to the passed length. Function returns the actual number of bytes read in the buffer, which caller can use later on. The function returns MPAC_FAILURE, if p_file_name contains NULL or buffer contains NULL or buff_size is negative or the system fails to open the file.

Return Value: Number of bytes read in buffer or MPAC_FAILURE.

Parameters: *p_file_name* Name of the file.
buffer Buffer in which file is to be read.
buff_size Size of the buffer.

2.3 int mpac_io_file_readln (int fd, char* buffer, int buff_size)

This function reads a line from the file specified by socket fd and reads it in the buffer. If the function fails to read from the file then it returns MPAC_FAILURE. If the EOF is reached during reading from the file then it returns MPAC_SUCCESS. If it successfully reads a line then the function returns the number of bytes read.

Return Value: MPAC_SUCCESS or MPAC_FAILURE or the number of bytes read.

Parameters: *fd* File descriptor to read from a file.
buffer Buffer in which file is to be read.
buff_size Size of the buffer.

3. mpac_io_net.h

Socket based network utility functions.

This file contains the prototype of the functions which can be used to get the IP address, domain name, port number and the proto number by giving some simple parameters. It also contains an initialization function which can be used to initialize the network handler *mpac_io_net_handler*.

Authors: Ghulam Mustafa, Umair Naushad

3.1 struct mpac_io_net_handler

A network handler that contains information about the communication type, protocol type, port number, message size, buffer size etc.

Members

- int side
- int fd
- int msg_size
- unsigned int sock
- unsigned int bytes_read
- int accept fd
- struct sockaddr in * recv_host
- unsigned long buff_size
- char* buffer
- struct sockaddr accept_sock
- socklen_t accept_sock_sz

int side

Sender or Receiver.

int fd

Main file descriptor.

int msg_size

Size of message.

unsigned int sock

Protocol type (TCP, UDP, RAW).

unsigned int bytes_read

Number of received bytes (for receive side only).

int accept_fd

File descriptor for accept() function call.

struct sockaddr in * recv_host

Host address.

unsigned long buff_size

Size of buffer.

char* buffer

Buffer to store message.

struct sockaddr accept_sock

Socket address(for receive side only).

socklen_t accept_sock_sz

Length of socket address (unassigned int).

3.2. int mpac_io_net_init (struct mpac_io_net_handler* h, const char* hostname, int port, const char* protocol, int cside, int msgsize, unsigned long buff_size)

This function will initialize the network handler // bf h according to the provide information about host name, protocol, port number, message size, buffer size and communication side.

Return Value:

Parameters: *h* A network handler. hostname Name of the host.
port Port number.
protocol Name of the protocol
cside Communication side (sender/receiver).
msgsize Message size.
buff_size Buffer size.

3.3 char* mpac_io_net_getipaddr (char* domainName)

This function receives the domainName as a character pointer and returns the corresponding IP address as a character pointer. If any error occurs while accessing the IP address or if the passed domainName is not valid then the function returns NULL.

Return Value: IP address or NULL.

Parameters: *h* A network handler.

3.4 int mpac_io_net_getportno (char* service, char* proto)

This function will returns port number according to given specification or MPAC_FAILURE in case of any error.

Return Value: Port number or MPAC_FAILURE.

Parameters: *service* Name of the used service.
proto Name of the used proto.

3.5 int mpac_io_net_getproto (char* protoName)

This function will return proto number according to given specification or MPAC_FAILURE in case of any error.

Return Value: Proto number or MPAC_FAILURE.

Parameters: *protoName* Name of the used proto.

3.6 char* mpac_io_net_getdn (char* ipAddress)

This function receives the ipAddress as a character pointer and returns the corresponding domain name as a character pointer. If any error occurs while accessing the domain name or if the passed IP address is not valid then the function returns NULL.

Return Value: Domain name or NULL.

Parameters: *h* A network handler.

4. mpac_io_tcp.h

TCP stream socket based network I/O services.

This file contains the prototype of the functions which are used to send and receive data through Transmission Control Protocol. By using these functions the user can start socket based network programming without understanding the underlying system calls like `socket()`, `setsockopt()`, `bind()`, `listen()`, `accept()`, `read()` and `write()`. To send data, by using the MPAC library, the user needs to use only two functions `mpac_io_tcp_open_sender()` and `mpac_io_tcp_send()`. Similar functions to receive data are `mpac_io_tcp_open_receiver()`, `mpac_io_tcp_accept_receiver()` and `mpac_io_tcp_receive()`. One final function which is used by both the sender and receiver is `mpac_io_tcp_close()`.

Authors: Ghulam Mustafa, Abdul Waheed

4.1 `int mpac_io_tcp_open_sender (struct mpac_io_net_handler* h)`

This function calls the `socket()` function to obtain a file descriptor and assign it to `h->fd`. After initialization of `h->fd`, it calls `setsockopt()` function to adjust the send and receive buffers. Finally, it connects to a remote host, which is specified in `h->recv_host`, by calling `connect()` function. Function returns `MPAC_SUCCESS`, if it successfully initializes the `h->fd`, buffers and connects to the remote host. Function returns `MPAC_FAILURE`, if the underlying functions fail to obtain a file descriptor or fail to connect to the host or to initialize the buffers or if the passed parameter `h` contains null.

Return Value: `MPAC_FAILURE` or `MPAC_SUCCESS`.

Parameters: *h* A network handler.

4.2 `int mpac_io_tcp_send (struct mpac_io_net_handler* h)`

This function is used to send data to the destination. The data to be sent is contained in `h->buffer`. After successfully sending the data, the number of bytes sent is saved in `h->bytes_read`. If the underlying `write()` function successfully sends the data, then the function returns `MPAC_SUCCESS`. If any error occurs while sending the data or the passed handler `h` contains null, then the function returns `MPAC_FAILURE`.

Return Value: `MPAC_FAILURE` or `MPAC_SUCCESS`.

Parameters: *h* A network handler.

4.3 int mpac_io_tcp_open_receiver (struct mpac_io_net_handler* h)

This function calls the socket() function to obtain a file descriptor and assign in to h->fd. It also binds the receiver to the port specified in h->recv_host->sin port by using the bind() function and then listen to the incoming connections by using the listen() function. Finally, it calls setsockopt() function to set the send and receive buffers. Function returns MPAC_SUCCESS, if all the above operations are completed successfully . Function returns MPAC_FAILURE, if the underlying function calls fail to obtain a file descriptor, to bind a port, to listen to the connections, to initialize the buffers or if the passed parameter h contains null.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

4.4 int mpac_io_tcp_accept_receiver (struct mpac_io_net_handler* h)

This function calls the accept() function to wait for any incoming connection request. The function remains in a blocking state unless it found a request. When it receives a request, it creates a new file descriptor to handle the request and assign it to h->accept fd. After creating this new socket(file descriptor), it again goes to the blocking state and waits for a new request. The newly created socket is ready to receive. Function returns MPAC_SUCCESS, if it successfully creates a new socket. Function returns MPAC_FAILURE, if the underlying function calls fail to obtain a new file descriptor or if the passed parameter h contains null.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

4.5 int mpac_io_tcp_receive (struct mpac_io_net_handler* h)

This function is used to receive data from the sender. The received data is contained in h->bu . After successfully receiving all the data the number of bytes received is saved in h->bytes read. If the underlying read() function successfully received the data, then the function returns MPAC_SUCCESS. If any error occurs while receiving the data or the passed handler h contains null, then the function returns MPAC_FAILURE.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

4.6 int mpac_io_tcp_close (struct mpac_io_net_handler* h)

This function is used to close the file descriptor by calling the shutdown() function. It first checks whether the passed argument *h* is used for sender or receiver and then closes the descriptor accordingly. If the underlying shutdown() function successfully closed the file descriptor, then the function returns MPAC_SUCCESS. If any error occurs while closing the file descriptor or the passed handler *h* contains null, then the function returns MPAC_FAILURE.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

5. mpac_io_udp.h

UDP stream socket based network I/O services.

This file contains the prototype of the functions which are used to send and receive data through User Datagram Protocol. By using these functions the user can start communication without understanding the underlying system calls like `socket()`, `setsockopt()`, `bind()`, `sendto()`, `recvfrom()` and `close()`. To send data the user needs to use only two functions `mpac_io_udp_open_sender()` and `mpac_io_udp_send()`. Similar functions to receive data are `mpac_io_udp_open_receiver()` and `mpac_io_udp_receive()`. One final function which is used by both the sender and the receiver is `mpac_io_udp_close()`.

Authors: Ghulam Mustafa, Umair Naushad

5.1 int mpac_io_udp_open_sender (struct mpac_io_net_handler* h)

This function calls the `socket()` function to obtain a file descriptor and assign in to `h->fd`. After initialization of `h->fd`, it calls `setsockopt()` function to adjust the send and receive buffers. Function returns `MPAC_SUCCESS`, if it successfully initializes the `h->fd` and buffers. Function returns `MPAC_FAILURE`, if the underlying functions call fail to obtain a file descriptor or to initialize the buffers or if the passed parameter `h` contains null.

Return Value: `MPAC_FAILURE` or `MPAC_SUCCESS`.

Parameters: *h* A network handler.

5.2 int mpac_io_udp_send (struct mpac_io_net_handler* h)

This function is used to send data to the destination. The data to be send is contained in `h->buffer` and the address of the destination is contained in `h->recv_host`. If the underlying `sendto()` function successfully sends the data, then the function returns `MPAC_SUCCESS`. If any error occurs while sending the data or the passed handler `h` contains null, then he function returns `MPAC_FAILURE`.

Return Value: `MPAC_FAILURE` or `MPAC_SUCCESS`.

Parameters: *h* A network handler.

5.3 int mpac_io_udp_open_receiver (struct mpac_io_net_handler* h)

This function calls the socket() function to obtain a file descriptor and assign in to h->fd. After initialization of h-fd, it calls setsockopt() function to set the send and receive buffers. It also binds the receiver to the port specified in h->recv_host->sin port. Function returns MPAC_SUCCESS, if it successfully initializes the h->fd and buffers. Function returns MPAC_FAILURE, if the underlying function calls fail to obtain a file descriptor or to initialize the buffers or if the passed parameter h contains null or if the bind() function fails.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

5.4 int mpac_io_udp_receive (struct mpac_io_net_handler* h)

This function is used to receive data from the sender. The received data is contained in h->bu and the address of the sender is contained in h->recv_host. After successfully receiving the data the number of bytes received is saved in h->bytes read. If the underlying recvfrom() function successfully received the data, then the function returns MPAC_SUCCESS. If any error occurs while receiving the data or the passed handler h contains null, then the function returns MPAC_FAILURE.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

5.5 int mpac_io_udp_close (struct mpac_io_net_handler* h)

This function is used to close the file descriptor by calling the close() function. If the underlying close() function successfully closed the file descriptor, then the function returns MPAC_SUCCESS. If any error occurs while closing the file descriptor or the passed handler h contains null, then the function returns MPAC_FAILURE.

Return Value: MPAC_FAILURE or MPAC_SUCCESS.

Parameters: *h* A network handler.

6. mpac_resolution.h

Time resolution related utility functions.

The functions in this file can be used to find out the resolution of C library functions clock_gettime() and gettimeofday().

NOTE: As clock * functions are in the real time library so we need -lrt in our linker statement.

Author: Ghulam Mustafa

6.1 struct mpac_resval

Structure to hold the data related to resolution.

Members

- double adiff
- int itr

double adiff

Average difference in microsecond.

int itr

Number of iterations required.

6.2 int mpac_resolution_gtd (struct mpac_resval* rv, int nd)

Tests the resolution of gettimeofday(). Sets the rv->adiff in microseconds and rv->itr to the number of calls required to get nd differences. Returns 0 if successful and -1 if failed.

Return Value: 0 or -1.

Parameters: *rv* Structure to hold data related to resolution.
 nd Number of differences.

6.3 int mpac_resolution_rtc (struct mpac_resval* rv, int nd)

Tests the resolution of real time clock i.e. clock_gettime(CLOCK_REALTIME, -). Sets the rv->adiff in microseconds and rv->it to the number of calls made to get nd differences. Returns 0 if successful and -1 if failed.

Return Value: 0 or -1.

Parameters: *rv* Structure to hold data related to resolution.
 nd Number of differences.

6.4 void mpac_resolution_rtc_com (void)

Displays the system wide real time clock resolution.

Return Value: void.

Parameters: void.

7. mpac_stat.h

Statistics related utility functions.

This file contains functions that can be used to find out statistics terms like mean, median, mode, variance, standard deviation, maximum and minimum values of a given data.

NOTE: As these functions uses math library so we need to add `-lm` in our linker command.

Author: Ghulam Mustafa

7.1 double mpac_stats_mode (double* data, int n)

Calculate mode of the passed data. If size n is negative then it returns -1.

Return Value: Mode or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.2 double mpac_stats_mean (double* data, int n)

Calculate arithmetic mean of the passed data. If size n is negative then it returns -1.

Return Value: Mean or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.3 double mpac_stats_median (double* data, int n)

Calculate median of the passed data. If size n is negative then it returns -1.

Return Value: Median or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.4 double mpac_stats_min (double* data, int n)

Find out the minimum value from the passed data. If size n is negative then it returns -1.

Return Value: Minimum value or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.5 double mpac_stats_max (double* data, int n)

Find out the maximum value from the passed data. If size n is negative then it returns -1.

Return Value: Maximum value or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.6 double mpac_stats_var (double* data, int n)

Calculate variance of the passed data. If size n is negative then it returns -1.

Return Value: Variance or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.7 double mpac_stats_std (double* data, int n)

Calculate standard deviation of the passed data. If size n is negative then it returns -1.

Return Value: Standard deviation or -1.

Parameters: *data* Array of data.
 n Size of array data.

7.8 int mpac_stats_conf_intr (double* ci, double* data, int n, double z)

Calculate confidence interval of the passed data. If size n is negative then it returns -1.

Return Value: -1 or 0.

Parameters: *ci* Confidence interval.
data Array of data.
n Size of array data.
z Level of confidence.

7.9 int mpac_stats_cmp (const void* a, const void* b)

If first argument is greater than the second function returns 1, if the first argument is less than the second function returns -1 and if both the arguments are equal function returns 0.

Return Value: 1, -1 or 0.

Parameters: *a* First value.
b Second value.

8. mpac_thread_manager.h

Thread manager for fork-and-join execution of non-interacting workers.

Author: Abdul Waheed, Ghulam Mustafa

8.1 struct mpac_thread_manager_t

Encapsulates the thread management related information. A pointer to an object of this type is passed to every thread management function as a handle.

Members

- int num_threads
- int num_live_threads
- pthread_t** worker_thr
- pthread_attr_t* w_attr
- unsigned int affinity
- void* (*work_routine) (void*)
- void** arg

int num threads

Total number of threads.

int num live threads

Number of live threads.

pthread_t worker_thr**

Acts as a handle for the new thread.

pthread_attr_t* w_attr

Specify which thread attribute to use.

unsigned int affinity

Affinity support flag.

void* (*work routine) (void*)

For consistency purposes, arg must be an array of primitive types or user defined types even if the same parameters are to be passed to all threads. This reduces synchronization overhead, too.

void arg**

Arguments passed to a thread.

8.2 struct mpac_lock_t

Encapsulates locking mechanism for collaborative work. It should be used for long-term waiting, provided that the workers are cancelable. For short-term locking, simply use mutexes/semaphores.

Members

- pthread_mutex_t lock
- pthread_cond_t go
- int nworker
- int narrived

pthread_mutex_t lock

Mutex lock object.

pthread_cond_t go

Mutex conditional object.

int nworker

Number of workers.

int narrived

Number of arrived threads.

8.3 int mpac_thread_manager_init (struct mpac_thread_manager_t* handle, int num workers, pthread_attr_t* w_attr, unsigned int aff , void* (*work_routine)(void*), void** arg)

This function takes the number of workers, worker routine, worker attribute structure (or NULL for default attributes) and pointer to an arg structure. It will copy the thread setup related information in the handle, which will be used in subsequent calls to the thread manager.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.
num workers Number of workers.
w_attr Specify which thread attribute to use.
aff Affinity support flag.
work_routine An array of data types.
arg Arguments passed to a thread.

8.4 int mpac_thread_manager_fork (struct mpac_thread_manager_t* handle)

This function creates the given No. of threads and let them do their work in parallel. It starts the created threads. It will create the requested number of threads with each thread receiving pointer to args structure.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.

8.5 int mpac_thread_manager_wait (struct mpac_thread_manager_t* handle)

It encapsulates the join operation for multiple threads.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.

8.6 int mpac_thread_manager_startj (struct mpac_thread_manager_t* handle, int num workers, pthread_attr_t* W_attr, unsigned int aff , void* (*work routine)(void*), void** arg)

It is a high level function that manages the lifecycle of requested number of threads

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.
num workers Number of workers.
w_attr Specify which thread attribute to use.
aff Affinity support flag.
work_routine An array of data types.
arg Arguments passed to a thread.

8.7 int mpac_thread_manager_startd (struct mpac_thread_manager_t* handle, int num workers, pthread_attr_t* w_attr, unsigned int a , void* (*work routine)(void*), void** arg)

It is a high level function that creates requested number of threads but does not wait for the joining i.e detach.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.
num workers Number of workers.
w_attr Specify which thread attribute to use.
aff Affinity support flag.
work_routine An array of data types.
arg Arguments passed to a thread.

8.8 int mpac_thread_manager_end (struct mpac_thread_manager_t* handle)

This function forces the worker threads specified by the handle to be killed and cleans up the thread manager. There should be no need to call this function except for the case when worker threads are blocked and a monitor thread could take this action. It encapsulates cancel operation for multiple threads.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.

8.9 int mpac_thread_manager_isolate (struct mpac_thread_manager_t* handle)

It encapsulates the detach operation for multiple threads and let them release their resources themselves. It takes ids of the n threads to be isolated.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.

8.10 int mpac_thread_manager_sendsig (struct mpac_thread_manager_t* handle, int sig)

It encapsulates the kill operation for multiple threads and sends a particular signal to all threads.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *handle* Contains thread management-related information.
sig Signal number.

8.11 void mpac_thread_manager_initlock (struct mpac_lock_t* ml, int nwrkr)

It initializes mpac_lock_t.

Return Value: void.

Parameters: *ml* Encapsulates locking mechanism for collaborative work.
nwrkr Number of workers.

8.12 void mpac_thread_manager_destroylock (struct mpac_lock_t* ml)

It destroys mpac_lock_t.

Return Value: void.

Parameters: *ml* Encapsulates locking mechanism for collaborative work.

8.13 int mpac_thread_manager_barrier (double* btime)

It is a high level function. It ensures that all threads start work at the same time and sets the barrier release time in time attribute of mpac_lock_t struct

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: *btime* Barrier release time.

8.14 int mpac_thread_manager_No_of_cores ()

This function returns the total number of cores in the system.

Return Value: Number of cores.

Parameters: void.

8.15 int mpac_thread_manager_core_affnity (pthread_t pid, int cpu)

Bind a thread to a particular core.

Return Value: MPAC_SUCCESS or MPAC_FAILURE.

Parameters: pid Thread id.
 cpu CPU number.

9.3 int mpac_time_frt_rtc (struct timespec* ts, void* (*func)(void*), void* arg)

It measures the running time of a function using clock_gettime(CLOCK_REALTIME, -). If it successfully measures time and sets it in the tv structure attribute then it returns 0 otherwise it returns -1.

Return Value: 0 or -1.

Parameters: *tv* Structure that contains time in seconds and micro seconds.
func A pointer to a function.
arg A pointer to function arguments.

9.4 double mpac_time_fcput (void* (*func)(void*), void* arg)

It measures the CPU time taken by a function. In case of any error returns -1.

Return Value: 0 or -1.

Parameters: *tv* Structure that contains time in seconds and micro seconds.
func A pointer to a function.
arg A pointer to function arguments.

9.5 double mpac_time_ggettime ()

This function returns the current system time as a double precision value in micro seconds.

Return Value: Time in micro seconds or MPAC_FAILURE.

Parameters: void.

9.6 double mpac_time_rgettime ()

This function returns the current system time as a double precision value in nano seconds.

Return Value: Time in nano seconds or MPAC_FAILURE.

Parameters: void.