**MacTech Article**
**Draft 3**
**11/20/95**

This long article was prepared for inclusion in a March 1996 MacTech article on multiprocessing. It was significantly abridged before appearing in print. It contains some useful pointers and tips on the use of the MP API.

## The MP API

DayStar Digital and Apple Computer have designed a multiprocessing, multitasking application programming interface, the 'MP API', for use with a new generation of PowerPC-based Mac OS multiprocessing systems. These systems are standard Macintoshes with one major exception: they contain more than one CPU. The MP API defines a set of services that allow developers to create and communicate with multiple elements of execution called 'tasks'. When tasks are run on a multiprocessor system they are scheduled and run simultaneously on all the available processors. On a four processor system it is not uncommon to see computationally intensive parts of applications executing three or more times faster after being modified to take advantage of the MP API.

## Advantages of the MP API

Task creation is accomplished by providing a pointer to a function already defined within existing application code. The most obvious advantage of this approach is that you can use existing tools and build processes to construct an MP aware application. No special compilers or packaging of the task code are required.

Tasks have complete access to all the memory in the system. If an application has retrieved and prepared data for processing it can simply tell the tasks where the data is. It is not necessary to move any data to specialized task-only memory thus avoiding expensive transactions over system busses.

According to the Apple/DayStar MP API specification the processors in an MP system must be cache-coherent. This means that the developer need not be concerned with the possibility that data stored in the cache of one processor has not yet been written to main memory. If any other processor accesses that memory the MP hardware will automatically ensure that the value cached within the other processor is retrieved rather than the value in main memory. The MP API's assumption of cache-coherency makes programming significantly easier; programming *non*-cache-coherent systems is far more error-prone and is not for the faint of heart.

Tasks run pre-emptively on all systems, including those with a single processor. If an application is willing to require the presence of PowerPC hardware and the shared library that provides the MP API services, the creation of MP aware applications can be greatly simplified. The application simply creates tasks and distributes the work accordingly. The tasks created could do all the work while the application checks for user events and controls the flow of data.

The MP API is Apple system software. It will be carried forward into Copland and is in fact a subset of the Copland tasking model. Investment in this architecture today is an investment in the future.

## Restrictions of the MP API

Tasks created using the MP API have two restrictions. First, the function that a task calls and all the functions that it in turn calls must not contain any 68K code. 68K multiple processing is not possible will never be supported by the MP API.

Second, tasks must not directly or indirectly call the Macintosh Toolbox. Not only does the Mac Toolbox still contain significant amounts of 68K code but, more importantly, it is not reentrant. That is the Toolbox is not designed to be accessed from more than one task simultaneously. For example, two tasks calling NewPtr() at the same time would almost certainly corrupt the heap from which they are allocating the memory. Calls to non-reentrant library functions must also be avoided.

Tasks should also be aware that they may be executing at any time, including when applications that did not create them are running. For this reason tasks should not try to make use of low-memory globals.

## Using the MP API

Multiprocessing should be used in areas of your application that could benefit from faster execution and/or user responsiveness. Developers often isolate the compute intensive code in their programs both for acceleration and cross-platform purposes. It is areas like these that usually provide the best candidates for MP acceleration. It should be noted that although any number of tasks can be created, in order to gain the maximum possible benefit from a multiprocessing machine at least one task per processor must be created.

### General MP Techniques

The techniques used to implement multiprocessing will of course depend on the type of work being done. Some of the ways that can be used to approach a given problem are described below, along with the types of problems for which they are most suitable.

Data Decomposition

Data decomposition involves creating a number of tasks that all do the same thing. The application distributes the work to be done as evenly as possible to the different tasks. Two approaches to distributing the work are described below.

*"Divide And Conquer"*

For functions that transform or generate well-defined, relatively small amounts of localized data the "Divide And Conquer" approach is normally the most appropriate.

This method involves creating a number of tasks and two message queues per task. Message queues are a feature of the MP API used for communication between tasks; they are discussed in more detail later. Each task uses one pair of queues -one for receiving work requests and one for posting results. The application splits the input and output data evenly amongst the tasks and posts a work request defining the work a task is expected to perform to each task's work request queue.

Tasks wait on their respective work request queues. When a request arrives the task performs the required work and posts a result to its results queue indicating that it has finished.

The main application can either return to its main event loop and check for completion later or, if the tasks are fast enough just wait for all the tasks to finish before proceeding. The application could also help out with part of the work itself. Generalizing the latter can be a useful technique for handling environments in which the MP API services are not available.

The "Divide And Conquer" approach is appropriate for operations like filtering, composition, resampling, and other transformations that can easily split the input and/or output data into computationally balanced pieces.

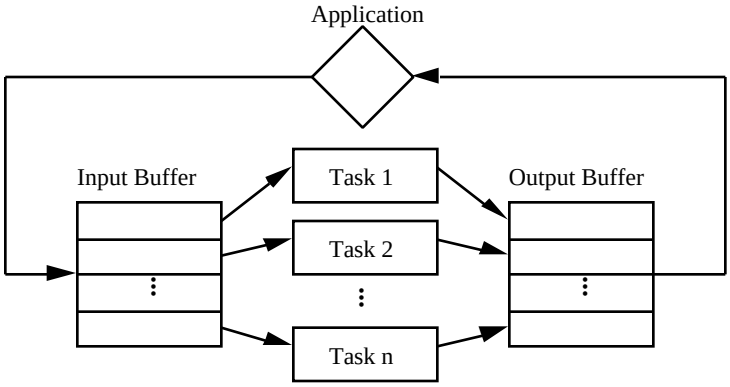


Fig.1 Data Decomposition: Divide And Conquer

*"Bank Line"*

For functions that use relatively large amounts of distributed input data to generate small amounts of output data in non-deterministic amounts of time the "Bank Line" approach is usually more appropriate. The analogy made here is with a waiting line at a bank. Several tellers are available to help customers but there is only one line in which customers stand. Customers receive help in the order in which they entered the line and take as long as necessary to complete their individual transactions.

In an application this method involves creating a number of tasks and only two message queues -a work request queue and a results queue. The application posts the work requests necessary to complete a job to the work request queue.

Tasks compete for messages from the work request queue. When a task obtains a work request it performs the work described and posts the result to the results queue. The task then starts all over again looking for another work request in the work request queue.

The task that will get a specific work request is non-deterministic. By the same token the order in which results are posted to the results queue is also non-deterministic. Since the application cannot determine the order in which results will be posted it is often convenient to have the tasks embed the results within the original work request and to post the modified work request to the results queue.

The application receives the results generated by the tasks via the results queue. As in the divide and Conquer" technique, the application can be checking events, controlling data flows and even be helping perform some of the work itself while the tasks are running.

The "Bank Line" technique is usually employed by rendering operations like ray tracing, and other types of functions that cannot predetermine what data that will be used, when requests for work will become available, or how long it will take to service requests.

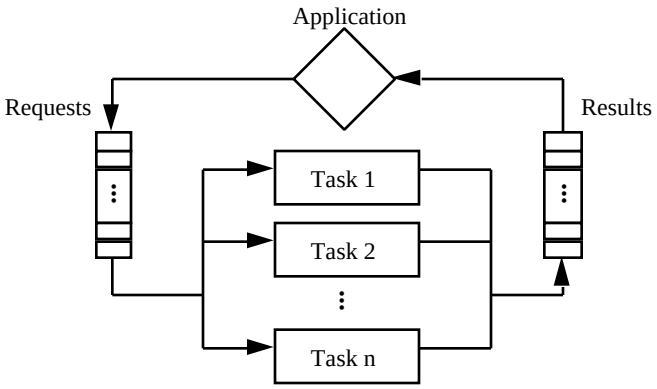


Fig. 2 Data Decomposition: Bank Line

Functional Decomposition

Functional decomposition involves creating a number of tasks that do different things. For example, a photo decompression operation may involve an initial decompression pass, followed by a color conversion pass, followed by a color correction pass, followed by an edge sharpening pass. The application creates tasks to work on the different parts of the operation. Tasks receive their input from the output of the task or tasks handling the preceding stage.

Functional decomposition is appropriate for operations that cannot decompose the data they are working with, or for applications that are concerned with user responsiveness during long, complicated operations. However, this technique has a significant drawback with regard to overall performance. Since the different stages of an operation will often have different processing requirements functional decomposition may often leave tasks idle waiting for more input from the preceding stage. If data decomposition can be used at any stage of a functional decomposition, or instead of a functional decomposition, then for the sake of overall performance it should be.

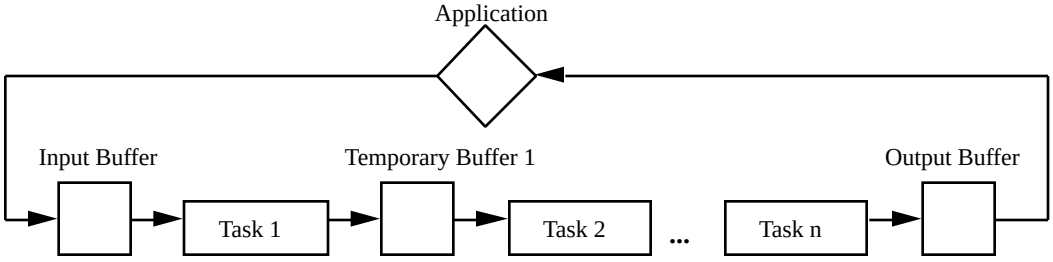


Fig. 3 Functional Decomposition

## MP API Services

The MP API document defines the set of services available to developers. Information about how to obtain the API is provided later. This section describes the general services available and touches on some of the important points to remember when using them.

Task Creation And Maintenance

Tasks are created using MPCreateTask(). The task entry point must be specified. It is usually a pointer to a function in your code and has the following prototype:

```
OSStatus fTask( void *parameter );
```

The parameter that the task will receive is also specified in the call to MPCreateTask(). It can be anything you like, a block of memory, a message queue, a C++ object, etc.

Tasks can be terminated by calling MPTerminateTask() or by simply letting the task return from its entry point. A task can choose to terminate at any time itself by calling MPExit(). Message queue based notification services are available to receive result codes from terminated tasks.

## Communication and Synchronization

Even though tasks and applications share the same memory it is very important that they communicate, at least initially, via one of the three provided communication primitives: message queues, semaphores and/or critical regions. Communicating via these primitives ensures that all former memory accesses made by the communicant are completed before the recipient starts using those locations. It also ensures that shared resources are accessed atomically, e.g. the "Bank Line" work request and result queues must be accessed by only one task at a time, otherwise the queue data structures would become inconsistent. Using the communication primitives also provides a method by which a task can yield time if it has to wait for something that is not yet available.

A very important point to remember is that the time required to perform the work in a given request should be much more than the amount of time it takes to communicate the request and its results. Losing sight of this need is the most common reason for a lack of good MP performance. As a rule of thumb try to consume at *least* a few milliseconds per request.

### *Message Queues*

Message queues are first in first out queues of 96 bit messages. Messages are useful for telling a task what work to do and where to look for information relevant to the request being made. They are also useful for indicating that a given request has been processed and if necessary what the results are.

Message queues incur more overhead than the other two communication primitives. If you cannot avoid frequent synchronization at least try to use a semaphore instead of a message queue.

### *Semaphores*

Semaphores store a value between zero and some arbitrary integer value greater than zero. The value in a semaphore can be raised and lowered, but never below zero and never above the semaphore's maximum value. Semaphores are useful for keeping track of how many occurrences of a particular thing are available for use.

Binary semaphores which have a maximum value of one are especially efficient mechanisms for indicating to some other task that something is ready. When a task or application has finished preparing data at some pre-agreed to location, it raises the value of a binary semaphore which the target task can be waiting on. The target task lowers the value of the semaphore, performs any necessary processing, and raises the value of a different binary semaphore to indicate that it is through with the data. This technique can be used to replace the message queue pairs described above in the "Divide And Conquer" technique. MPCreateBinarySemaphore() is a macro that exists to simplify the creation of binary semaphores.

*Critical Regions*

Critical regions are used to ensure that no more than one task (or the application) is executing a given 'region' of code at any given time. For example, if part of a task's job is to search a tree and modify it before proceeding with its primary work, then if multiple tasks were allowed to search and try to modify the tree at the same time, the tree would quickly become corrupted. An easy way to avoid the problem is to form a critical region around the tree searching and modification code. When a task tries to enter the critical region it will only be able to do so if no other task is currently in it thus preserving the integrity of the tree.

Critical regions should be used carefully. They introduce forced linearity into task execution which can be bad for performance. For example, if the tree search described above constituted the bulk of a task's work then the tasks would spend most of their time just trying to get permission to enter the critical region at great cost to overall performance. A better approach in this case might be to use different critical regions to protect subtrees. It would then be possible for one task to search one part of the tree while others were simultaneously working on other parts of the tree.

Memory Routines

Since the ability to allocate memory is so important to so many different kinds of functions, memory services that allocate, copy and free memory are available in the MP API .The memory is obtained from the heap of the application that created the task.

Task Scheduling

All tasks created using the MP API are initially placed at the end of a queue of tasks waiting to be run. As processors become available tasks are removed from the head of the run queue and assigned to them. Tasks run preemptively. That is, they are given a certain amount of time to run on a processor and at the end of that time they are interrupted and returned to the end of the run queue. This ensures that all the tasks in the system get fair access to the available processors.

Whenever a task specifies a duration of kDurationForever while waiting on one of the communication primitives and the primitive is not available the task will automatically release the processor it was using and add itself to the list of tasks waiting on the primitive. Waiting tasks will not be resumed, nor will they incur any overhead until the primitive on which they are waiting becomes available.

Sometimes a task may find it convenient to use a duration of kDurationImmediate when accessing a communication primitive. If nothing is available the task might then do some other work after which it would come back and check the primitive again. This will have the effect of ensuring that the task uses as much processor time as it possibly can which may unnecessarily impact the performance of other, more useful tasks. To alleviate the situation the task could call MPYield() after checking the primitive. MPYield() allows a task to give up the processor it is on and to return itself to the end of the run queue thus allowing better throughput of other tasks in the system.

## An MP API Example

PowerFrax is a simple example of the "Divide And Conquer" approach used to generate fractals.

When PowerFrax initializes it calls fTasksCreate() which first checks for the presence of the MPLibrary. It then ensures that the header files being used are compatible with the library. If either condition is not satisfied then an error will be returned and PowerFrax proceeds as though the MPLibrary is not present. Note: it is important to 'weak link' to the MPLibrary when building an application otherwise it will not launch when the MPLibrary is not present.

```
/* Check that the MP library is present */
if( theErr == noErr )
      if( !MPLibraryIsLoaded() )
            theErr = 1;

/* Check that the library is compatible with the headers being    */
/* used.                                                          */
if( theErr == noErr )
      if( !MPLibraryIsCompatible() )
            theErr = 1;
```

Next, fTasksCreate() calculates a task count. PowerFrax modifies this count according to a parameter. Applications will generally not implement this level of control.

```
/* Get the processor count. Note that the host processor is       */
/* included.                                                      */
if( theErr == noErr ) {
      gNumProcessors = MPProcessors();
      if( howMany < 1 )
            gNumTasks = gNumProcessors;
      else
            gNumTasks = howMany;
      }
```

Next, fTasksCreate() allocates a unique block of data for each task. PowerFrax uses the data blocks to store information that the tasks need to function correctly.

```
/* Create an array of sDataPtr (defined above) with one entry for */
/* each task to be created.                                       */
gTaskData = NULL;
if( theErr == noErr ) {
      gTaskData = (sTaskDataPtr)NewPtrClear(
            gNumTasks * sizeof( sTaskData ) );
      if( gTaskData == NULL )
            theErr = MemError();
      }
```

A task and two queues are created for each processor. The IDs of the task and queues are stored in the task's data block. The first parameter to MPCreateTask() is the function to be used for the task: fTask(). The second parameter is the parameter that will be passed to fTask(). In this case it is a pointer to the task's data block.

```
/* Create one task per processor and leave them active until the  */
/* application quits.                                             */
for( i = 0; i < gNumTasks && theErr == noErr; i++ ) {
      /* Create the message queues by which to communicate the    */
      /* action messages the app. sends to the tasks, and by      */
      /* which to receive the status message returned by the      */
      /* tasks.                                                   */
      if( theErr == noErr )
            theErr = MPCreateQueue( &gTaskData[i].appToTask );
      if( theErr == noErr )
            theErr = MPCreateQueue( &gTaskData[i].taskToApp );
      if( theErr == noErr )
            theErr = MPCreateTask( fTask, &gTaskData[i],
                  2048, NULL, NULL, NULL, 0, &gTaskData[i].taskID );
```

```
            if( theErr == noErr ) {
                  fSendMessage( gTaskData[i].appToTask, kTMCreate );
                  fReceiveMessage( gTaskData[i].taskToApp, &message );
                  }
            }
```

The initialization process is now complete. The tasks will be waiting for their first work requests.

When PowerFrax decides to generate an image it asks each task to calculate one scan line worth of data. To make a work request PowerFrax records the parameters defining the line to be calculated in a task's data block along with a pointer to the buffer into which to place the results; in this case a pointer directly into a GWorld pixmap. A request is then posted to the task's request queue.

```
void fTaskRun( long whichTask, double *zc, double *zd, double *step,
      double *escape, short width, char *results ) {
/* Start the specified task working on the specified scanline.          */

      gTaskData[whichTask].zc = *zc;
      gTaskData[whichTask].zd = *zd;
      gTaskData[whichTask].step = *step;
      gTaskData[whichTask].escape = *escape;
      gTaskData[whichTask].width = width;
      gTaskData[whichTask].results = results;

      fSendMessage( gTaskData[whichTask].appToTask, kTMRun );
      }
```

The task, which is currently waiting on the 'appToTask' queue, or more appropriately the work request queue, gets the 'kTMRun' message (kTMRun is an application defined constant). It responds by using the parameters that were stored in its data block to calculate one scanline of data. It then sends a 'kTMReady' message to its result queue.

```
OSStatus fTask( void *theParameter ) {
/* This is the task.                                                    */

      Boolean finished;
      sTaskDataPtr p;
      long message;

      /* The following statement is executed once, immediately after    */
      /* task creation.                                                 */
      p = (sTaskDataPtr)theParameter;

      finished = false;
      while( !finished ) {
            fReceiveMessage( p->appToTask, &message );
            switch( message ) {
                  case kTMCreate:
                        break;
                  case kTMRun:
                        main( &p->zc, &p->zd, &p->step, &p->escape,
                              p->width, p->results );
                        break;
                  case kTMQuit:
                        finished = true;
                        break;
                  }
```

```
            fSendMessage( p->taskToApp, kTMReady );
            }

      return( noErr );
      }
```

After PowerFrax has started each of the tasks running, it waits for the 'kTMReady' message to appear in each of the result queues. If less than a tenth of a second has passed PowerFrax repeats the process, otherwise it displays the results to date and handles all pending user events. Once the events are handled PowerFrax continues calculating the fractal.

When PowerFrax quits it sends each task a message asking it to clean up. The only thing the task does is return from its entry point. PowerFrax then deletes the task's queues and terminates the task. The task data blocks are disposed of last.

```
void fTasksQuit( void ) {
/* This function is called when the application is quit. It notifies    */
/* each successfully created task to initiate an orderly shutdown and   */
/* then releases the resources allocated during task creation.          */

      long i;
      long message;

      if( gTaskData != NULL ) {
            for( i = 0; i < gNumTasks; i++ ) {
                  /* If this task was successfully created then try to  */
                  /* close it down properly.                            */
                  if( gTaskData[i].appToTask != NULL &&
                        gTaskData[i].taskToApp != NULL &&
                        gTaskData[i].taskID != NULL ) {
                        fSendMessage( gTaskData[i].appToTask, kTMQuit );
                        fReceiveMessage( gTaskData[i].taskToApp, &message );
                        }
                  /* Delete the task resources */
                  if( gTaskData[i].appToTask != NULL )
                        MPDeleteQueue( gTaskData[i].appToTask );
                  if( gTaskData[i].taskToApp != NULL )
                        MPDeleteQueue( gTaskData[i].taskToApp );
                  if( gTaskData[i].taskID != NULL )
                        MPTerminateTask( gTaskData[i].taskID, noErr );
                  }
            DisposePtr( (Ptr)gTaskData );
            }
      }
```

The functions fSendMessage() and fReceiveMessage() are just wrappers to the message passing functions MPNotifyQueue() and MPWaitOnQueue() respectively.

The actual gains seen in PowerFrax execution times are close to linear. With two processors fractals are generated almost twice as fast. With four processors the generation times are almost four times as fast. Applications will not always see this type of improvement. Since all processors must access the memory subsystem in a linear fashion, tasks that do not perform significant calculations on data will not perform as well as tasks that do.

### Adding MP API Support

The amount of time required to modify an application to be MP aware will of course vary greatly depending on how much functionality is targeted, and how well it has been isolated from the rest of the program and the Macintosh Toolbox. "Divide And Conquer" algorithms tend to be significantly easier to implement than "Bank Line" algorithms. PowerFrax was literally converted in an evening and has not changed substantially since. Real applications have seen major performance boosts gained in as little as one to three days with bugs discovered and fixed over longer intervals.

The cost of a DayStar Genesis MP system to the end user is around 10 to 15 thousand dollars. Each system comes with four high speed 604 PowerPC processors and a quad speed CD ROM. The amount of RAM and the disk drives that come with the machine will vary depending on the reseller from which the machine is purchased but will not be less than 16MB and 1GB respectively. DayStar offers special developer prices on the Genesis MP, on multiprocessor upgrade cards for Apple's 7500, 8500 and 9500 machines, and on our 040 and PPC upgrades in general.

To obtain sales information you can contact us at (770) 967 2077. We also maintain a World Wide Web site at www.daystar.com. This site contains more useful information for MP API developers including the latest SDK. The SDK contains extensive documentation, the shared library which implements the MP API on uniprocessor systems and the PowerFrax sample code featured in this article. To inquire about MP in general you can use the above number and leave a message at ext. 267 or send email to mp@daystar.com.