

# Simple C Multitasking Library for the Apple Macintosh Computer

Viktor K. Decyk  
Department of Physics and Astronomy  
UCLA  
Los Angeles, California 90095-1547  
decyk@physics.ucla.edu

The Apple Macintosh Operating System OS X has multiprocessing capabilities, which allows computers with multiple processors to perform multiple tasks simultaneously. The programming interface which currently supports multiprocessing is the Multiprocessing Services Library (v. 2.1) and is described in the document, "Multiprocessing Service Programming Guide." This document is available at the web address: [http://developer.apple.com/documentation/Carbon/Conceptual/Multitasking\\_MultiproServ/](http://developer.apple.com/documentation/Carbon/Conceptual/Multitasking_MultiproServ/). This programming interface is low-level and unnecessarily complex for simple tasks. We have therefore written a simpler interface, which we call `MacMP.c`, that addresses more directly the typical needs of scientific programmers. This library is freely available on our web site: <http://exodus.physics.ucla.edu/appleseed/> and has been tested with the gcc compiler

This simplified interface is similar in spirit to the simple Multitasking library on the Cray parallel-vector computers of a decade ago [1]. Multitasking is done at a procedure level, and our simple interface assumes that no two tasks will write to the same areas of memory. For example, suppose that we have a C procedure `push` for accelerating `NPMAX` particles in a given electric field, as follows:

```
#define NPMAX    20000
#define NXS      256
int np, nx;
float particles[NPMAX][2], efield[NXS], energy;
void push(float *pa, float *f, int *np, int *nx, float *ek);

np = NPMAX;
nx = NXS;
push(&particles[0][0], efield, &np, &nx, &energy);
```

where the variable `energy` contains the kinetic energy when the procedure returns. To take advantage of a second processor, observe that instead of a single call to the `push` procedure, one can call it twice, each time with `npmax/2` of the particles, as follows:

```
float energy2;

np = NPMAX/2;
push(&particles[0][0], efield, &np, &nx, &energy);
push(&particles[np][0], efield, &np, &nx, &energy2);
energy += energy2;
```

To prevent both procedures from writing to the same `energy` variable, we have

created an extra variable, `energy2`, which is added to the `energy` variable to obtain the total energy. The expression `particles[np][0]` means that the second procedure will start processing the particle array with particle number `np`, instead of with particle 0. The other arguments of `push` are input only and never modified.

If the computer contains two cpus, one can take advantage of the second cpu by launching the second procedure as a task. To do this, we must first initialize the Multiprocessing library, as follows:

```
#include "MacMP.h"

    int nproc;

    MP_Init(&nproc);
```

After execution, the variable `nproc` will contain the number of processors found (or zero if multiprocessing is not supported on this computer).

In order to launch the second procedure as a task, we execute the procedure `MP_Taskstart`, whose first three arguments are a taskid, the name of the procedure to be multitasked, and the number of arguments in the procedure, followed by the arguments, as follows:

```
    int idtask, nargs;
/* start task on second cpu */
    nargs = 5;
    MP_Taskstart(&idtask, &push, &nargs,
                &particles[np][0], efield, &np, &nx, &energy2);

/* give main cpu some work */
    push(&particles[0][0], efield, &np, &nx, &energy);

/* wait for second cpu to complete */
    MP_Taskwait(&idtask);

    energy += energy2;
```

If the task was successfully launched, the variable `idtask` will contain a non-zero value. One uses this `idtask` to wait for the task to complete by calling the procedure `MP_Taskwait`. In this example we are processing the second half of the particles as a task on the second cpu, then processing the first half of the particles on the first cpu, then we wait for the second cpu to complete before proceeding with any remaining calculation. When the multiprocessing is finished, one should call

```
    MP_End();
```

in order to terminate multiprocessing. That's it! Appendix A shows a simple multi-tasking version of the `push` procedure which hides the details of multiprocessing from the main program.

Each time `MP_Taskstart` is called, a new task is created, and it is terminated when

MP\_Taskwait returns. The overhead for starting a task is about 100 microseconds on a 2.0 GHz Macintosh G5, so that the tasks should have substantially more work than that to be worthwhile. If the same task will be called repeatedly in a loop, it is possible to reduce the overhead to about 35 microseconds by not creating a new task each time, but rather by sending the task a signal to start, and waiting for a signal back when it is done. To use this improved method, one first initializes the task (outside of the loop) rather than starts it:

```
MP_Taskinit(&idtask, &push, &nargs,
            &particles[np][0], efield, &np, &nx, &energy2);
```

and then starts the task inside the loop with the procedure MP\_Sndsig, as follows:

```
int i, ierr;

for (i = 0; i < 1000; i++) {
/* start task on second cpu */
    ierr = MP_Sndsig(&idtask);
/* give main cpu some work */
    push(particles, efield, &np, &nx, &energy);
/* wait for second cpu to complete */
    ierr = MP_Waitsig(&idtask);
    energy += energy2;
    ...
}
```

The procedure MP\_Waitsig waits for the task to send a signal back that it is finished. The next time through the loop, MP\_Sndsig will restart the task again. When the loop is finished, one should destroy the task as follows:

```
MP_Killtask(&idtask);
```

As before, one should call MP\_Init and MP\_End to initialize and terminate multiprocessing. One can initialize up to 16 tasks at one time. The number of tasks running at one time need not coincide with the number of cpus, but this is generally the most efficient use of multitasking. Appendix B shows a reusable multi-tasking version of the push procedure which hides the details of multiprocessing from the main program. To create an executable, the compile command is:

```
gcc -O -c -I /Developer/Headers/FlatCarbon MacMP.c
gcc -O main.c MacMP.o \
/System/Library/Frameworks/Carbon.framework/Carbon
```

There are certain restrictions on the procedures used in tasks. First of all, all task procedures should return void and all arguments should be pointers. Two tasks should not write to the same location in memory. (That is, tasks should be reentrant.) Tasks should not use temporary variables as actual arguments, such as return values of functions. If the task is started in one procedure but terminates in another, one must be careful about using variables which are local to the first procedure as arguments, because they may become undefined when this procedure terminates. Another important restriction is that the tasks cannot safely call the MacOS, directly or indirectly.

For most scientific programs, this mainly means that I/O (printf or fprintf statements) and memory allocations are not supported in the tasks. The total number of arguments in a task should not be greater than 24.

A number of procedures have been added to help in debugging multitasking codes which are not working properly. The first such procedure is `MP_Setstack`. Tasks by default are set to use a stack size of 16384 bytes. In some cases this is not large enough and the task will crash without warning. One can double the stack size, for example, by calling `MP_Setstack` before creating the task, for example:

```
MP_Setstack(32768);
```

If enlarging the stack size does not cure the problem, one can check to see if the arguments of the task are correct. To do this, we have added three additional procedures. The first of these is `MP_Taskbuild`. It creates a task exactly as `MP_Taskstart` (and has the same arguments), but does not run it. One can substitute `MP_Taskbuild` in place of `MP_Taskstart`. The second subroutine is `MP_Runtask`. This will manually run the task created by `MP_Taskbuild`, but runs it on the main cpu. As a result, one can add print statements into this task. `MP_Runtask` has the same arguments as `MP_Taskwait`, and can be substituted for it. Finally, there is the procedure `prparms`, whose argument is the taskid, which will print out the current arguments of the task. If the task works properly with `MP_Runtask`, but not with `MP_Taskstart`, then it is likely that the two tasks are writing to some common memory location, either directly or indirectly by some compiler construction.

## Acknowledgments

I wish to acknowledge help by Dean Dauger from UCLA and George Warner from Apple. This work has supported by NSF contracts DMS-9722121 and PHY 93-19198 and DOE contracts DE-FG03-98DP00211, DE-FG03-97ER25344, DE-FG03-86ER53225, and DE-FG03-92ER40727.

## References

[1] Multitasking User Guide, Cray Research, Inc. 1985.

## Appendix A

A sample multi-tasking version of the push procedure which encapsulates the multiprocessing.

```
void push(float *part, float *fx, int *nop, int *nx, float *ek);

void m2push(float *particles, float *efield, int *nop, int *nx,
            float *energy, int *ierr) {
#include "MacMP.h"
/* multitasking particle push for two processors */
/* local data */
    typedef void (*aProcPtr)();
    int nargs = 5, idtask, npp, npl;
    float energy2;
/* initialize constants */
    npp = *nop/2;
    npl = *nop - npp;
    idtask = 0;
    *ierr = 0;
/* start particle push task on second cpu */
    MP_Taskstart(&idtask, (aProcPtr)&push, &nargs,
                &particles[2*npp], efield, &npl, nx, &energy2)
/* check for errors */
    if (!idtask) {
        *ierr = -1;
        return;
    }
/* give main cpu some work */
    push(particles, efield, &npp, nx, energy);
/* wait for task to complete */
    MP_Taskwait(&idtask);
    if (idtask)
        *ierr = -2;
    *energy += energy2;
    return;
}
```

## Appendix B

A sample reusable multi-tasking version of the push procedure which encapsulates the multiprocessing.

```
void push(float *part, float *fx, int *nop, int *nx, float *ek);

void m2push(float *particles, float *efield, int *nop, int *nx,
            float *energy, int *ierr) {
#include "MacMP.h"
/* reusable multitasking particle push for two processors */
/* local data */
    typedef void (*aProcPtr)();
    static int first = 1, nargs = 5, idtask = 0, npp, npl;
    static float energy2;
/* initialize constants */
    npp = *nop/2;
    npl = *nop - npp;
    *ierr = 0;
/* initialize particle push task on first entry */
    if (first==1) {
        MP_Taskinit(&idtask, (aProcPtr)&push, &nargs,
                   &particles[2*npp], efield, &npl, nx, &energy2)
/* check for errors */
        if (!idtask) {
            *ierr = -1;
            return;
        }
        else
            first = 0;
    }
/* start task on second cpu */
    *ierr = MP_Sndsig(&idtask);
    if (*ierr)
        return;
/* give main cpu some work */
    push(particles, efield, &npp, nx, energy);
/* wait for task to complete */
    *ierr = MP_Waitsig(&idtask);
    *energy += energy2;
    return;
}
```