

Simple Fortran 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/. This programming interface is low-level and unnecessarily complex for simple tasks. We have therefore written a simpler interface, which we call `MacMP.f`, 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 Absoft, IBM, and Nag Fortran compilers.

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 subroutine 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 Fortran subroutine `push` for accelerating `npmax` particles in a given electric field, as follows:

```
parameter(npmax=20000,nx=256)
real particles(2,npmax), efield(nx), energy

call push(particles,efield,npmax,nx,energy)
```

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

```
integer np
real energy2

np = npmax/2
call push(particles,efield,np,nx,energy)
call push(particles(1,np+1),efield,np,nx,energy2)
energy = energy + energy2
```

To prevent both subroutines 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(1,np+1)` means that the second subroutine will start processing the particle array with particle number `np + 1`, instead of with particle 1. 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 subroutine as a task. To do this, we must first initialize the Multiprocessing library, as follows:

```
integer nproc  
  
call 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 subroutine as a task, we call the procedure `MP_TASKSTART`, whose first three arguments are a taskid, the name of the subroutine to be multitasked, and the number of arguments in the subroutine, followed by the arguments, as follows:

```
integer idtask, nargs  
c start task on second cpu  
  nargs = 5  
  call MP_TASKSTART(idtask,push,nargs,  
    &particles(1,np+1),efield,np,nx,energy2)  
  
c give main cpu some work  
  call push(particles,efield,np,nx,energy)  
  
c wait for second cpu to complete  
  call MP_TASKWAIT(idtask)  
  
  energy = 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

```
call MP_END()
```

in order to terminate multiprocessing. That's it! Appendix A shows a simple multitasking version of the `push` subroutine 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:

```
call MP_TASKINIT(idtask,push,nargs,  
&particles(1,np+1),efield,np,nx,energy2)
```

and then starts the task inside the loop with the procedure MP_SNDSIG, as follows:

```
integer i, ierr  
  
do 10 i = 1, 1000  
c start task on second cpu  
  ierr = MP_SNDSIG(idtask)  
c give main cpu some work  
  call push(particles,efield,np,nx,energy)  
c wait for second cpu to complete  
  ierr = MP_WAITSIG(idtask)  
  energy = energy + energy2  
  ...  
10 continue
```

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:

```
call 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 subroutine which hides the details of multiprocessing from the main program.

To create an executable with the Absoft Fortran compiler, the command is:

```
f77 -O -plainappl main.f MacMP.f
```

To create an executable with the IBM compiler, one needs to use a Fortran wrapper MacMPxlf.c to the C version of MacMP, as well as the header MacMP.h. The compile commands are:

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

To create an executable with the Nag compiler, one needs to use a Fortran wrapper MacMPf77.o to the C version of MacMP, as well as the header MacMP.h. The compile

commands are:

```
gcc -O -c MacMPf77.c
gcc -O -c -I /Developer/Headers/FlatCarbon MacMP.c
f95 -O main.f -framework carbon MacMPf77.o MacMP.o
```

There are certain restrictions on the procedures used in tasks. First of all, 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 subroutine but terminates in another, one must be careful about using variables which are local to the first subroutine as arguments, because they may become undefined when this subroutine 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 (print or write statements) are not supported in the tasks. The total number of arguments in a task should not be greater than 24. Finally, character variables should not be used as actual arguments.

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:

```
call 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 subroutine which encapsulates the multiprocessing.

```
      subroutine m2push(particles,efield,nop,nx,energy,ierr)
c multitasking particle push for two processors
      implicit none
      integer nop, nx, ierr
      real particles(2,nop), efield(nx), energy
c local data
      integer nargs, idtask, npp, npl
      real energy2
      external push
      data nargs /5/
c initialize constants
      npp = nop/2
      npl = nop - npp
      idtask = 0
      ierr = 0
c start particle push task on second cpu
      call MP_TASKSTART(idtask,push,nargs,particles(1,npp+1),
        &efield,npl,nx,energy2)
c check for errors
      if (idtask.eq.0) then
        ierr = -1
        return
      endif
c give main cpu some work
      call push(particles,efield,npp,nx,energy)
c wait for task to complete
      call MP_TASKWAIT(idtask)
      if (idtask.ne.0) ierr = -2
      energy = energy + energy2
      return
      end
```

Appendix B

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

```
      subroutine mx2push(particles,efield,nop,nx,energy,ierr)
c reusable multitasking particle push for two processors
      implicit none
      integer nop, nx, ierr
      real particles(2,nop), efield(nx), energy
c function declarations
      integer MP_SNDSIG, MP_WAITSIG
      external push
c local data
      integer first, nargs, idtask, npp, npl
      real energy2
      save first, nargs, idtask, npp, npl, energy2
      data first, nargs, idtask /1,5,0/
c initialize constants
      npp = nop/2
      npl = nop - npp
      ierr = 0
c initialize particle push task on first entry
      if (first.eq.1) then
          call MP_TASKINIT(idtask,push,nargs,particles(1,npp+1),
              &efield,npl,nx,energy2)
c check for errors
          if (idtask.eq.0) then
              ierr = -1
              return
          else
              first = 0
          endif
      endif
c start task on second cpu
      ierr = MP_SNDSIG(idtask)
      if (ierr.ne.0) return
c give main cpu some work
      call push(particles,efield,npp,nx,energy)
c wait for task to complete
      ierr = MP_WAITSIG(idtask)
      energy = energy + energy2
      return
      end
```