```
@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __init():
    # Create parallel closure (task manager) of the type
    # we are interested in (coarse grain parallel list manager) ...
    return spm.pclosure.macro.pinterp.list.grainCoarse.policyA.defun \
             (signature = 'is_prime::main', # Something unique to module.
              stage1Cb  = __taskStat,
             );

__pc      = __init();
__nprimes = 0;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOnline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def __taskStat(pc):
    # Callback for incoming status reports ...
    try:
        global __nprimes;

        hdl        = pc.stage1.payload.tie();
        __nprimes += hdl.spm.stat.returnValue;
        print('  --> Rolling count of (# of prime numbers) :: %d' \
              % (__nprimes,));
    except (SPMTaskDropped,
            SPMTaskLoad,
            SPMTaskEval,
            ), (hdl,):
        pass;

    return (pc.stage1.event.done(), # Explicitly let the backend
                                    # know we are done;
            None,
            )[-1];


#
# Compute the number of primes between 3 and 502347
# by dividing the range into 'nBuckets' ...
#
import os;

@spm.util.dassert(predicateCb = spm.sys.sstat.amOffline)
@spm.util.dassert(predicateCb = spm.sys.pstat.amHub)
def main(pool,
         nBuckets = 10,
         ):
    # Initialize 'stage0'.
    global __nprimes;

    assert(nBuckets >= 1);
    __pc.stage0.init.main(typedef = \
        r"""
         task<list>::struct {
           #
           # SPM component ...
           #
           spm::struct {
             meta::struct {
               label         ::scalar<stringSnippet> = deferred;
               path          ::tuple<string>         = deferred;
               modulePreload::tuple<string>          = deferred;
               module        ::scalar<stringSnippet> = deferred;
               timeout       ::scalar<timeout>       = deferred;
             };

             core::struct {
               relaunchPre  ::scalar<bool>           = None;
               relaunchPost ::scalar<bool>           = None;
             };

             stat::struct {
               exception    ::scalar<auto>           = None;
               returnValue  ::scalar<auto>           = None;
             };
           };

           #
           # non-SPM component ...
           #
           nMin             ::scalar<auto>           = deferred;
           nMax             ::scalar<auto>           = deferred;
         };
         """);
    __nprimes = 0;                          # Always reset counter.
    hdl       = __pc.stage0.payload.tie(); # Handle to the payload.
    nMin      = 2;
    for ct in range(0, nBuckets):
        # Initialize 'spm' component so that Spokes know what to
        # preload ...
        hdl.spm.meta.label         = '***';
        hdl.spm.meta.path          = \
             (os.path.dirname(__pc.meta.module.srcDir),);
        hdl.spm.meta.modulePreload = ('is_prime',);
        hdl.spm.meta.module        =  'is_prime';
        hdl.spm.meta.timeout       = \
             spm.util.timeout.after(seconds = 10);
        hdl.nMin                   = nMin; nMin += ((502347) / nBuckets);

        if (ct == (nBuckets - 1)):
            hdl.nMax = 502347;
        else:
            hdl.nMax = nMin;

        hdl.Push();

    #
    # Invoke the pmanager ...
    #
    __pc.stage0.event.manage \         (pool                  = pool,
         nSpokesMin           = spm.env.const.default,
         nSpokesMax           = spm.env.const.default,
         timeoutWaitForSpokes = spm.util.timeout.after(seconds = 2),
         timeoutExecution     = spm.util.timeout.after(seconds = 300),
        );

    return;
```