An Investigation of Symmetric Multi-Threading Parallelism for Scientific Applications

Installation and Performance Evaluation of ASCI sPPM

Frank Castaneda
fcastaneda@nc.rr.com

Nikola Vouk
nvouk@ncsu.edu

North Carolina State University
CSC 591c Spring 2003
May 2, 2003
Dr. Frank Mueller

An Investigation of Symmetric Multi-Threading Parallelism for Scientific Applications

Introduction

Our project is to investigate the use of Symmetric Multi-Threading (SMT), or "Hyper-threading" in Intel parlance, applied to course-grain parallelism in large-scale distributed scientific applications. The processors provide the capability to run two streams of instruction simultaneously to fully utilize all available functional units in the CPU. We are investigating the speedup available when running two threads on a single processor that uses different functional units.

The idea we propose is to utilize the hyper-thread for asynchronous communications activity to improve course-grain parallelism. The hypothesis is that there will be little contention for similar processor functional units when splitting the communications work from the computational work, thus allowing better parallelism and better exploiting the hyper-thread technology. We believe with minimal changes to the 2.5 Linux kernel we can achieve 25-50% speedup depending on the amount of communication by utilizing a hyper-thread aware scheduler and a custom communications API.

Experiment Setup

Software Setup	Hardware Setup
?? Custom Linux Kernel 2.5.68 with Red Hat distribution ?? Modification of Kernel Scheduler to run processes together ?? Custom Test Code	?? IBM xSeries 335 Single/Dual Processor 2.0 Ghz Xeon

In order to test our code we focus on the following scenarios:

?? A serial execution of communication / computation sections

- Executing in serial is used to test the expected back-to-back run-time. There is no busy wait in this code.

?? A SPMD execution of the serial program with each process doing ½ work of original on a SMT system

- We test how the processors react to the same program forked and each doing ½ the rounds of the whole program. It is expected to see major contention in the functional unit usage

?? A SPMD execution of the serial program with each process doing ½ work of original on a SMP system (Same system 2nd processor enabled and hyperthreads disabled)

- This test would represent the absolute maximum speed that can be achieved if there was no contention for shared functional units. (*In our ideal application*)

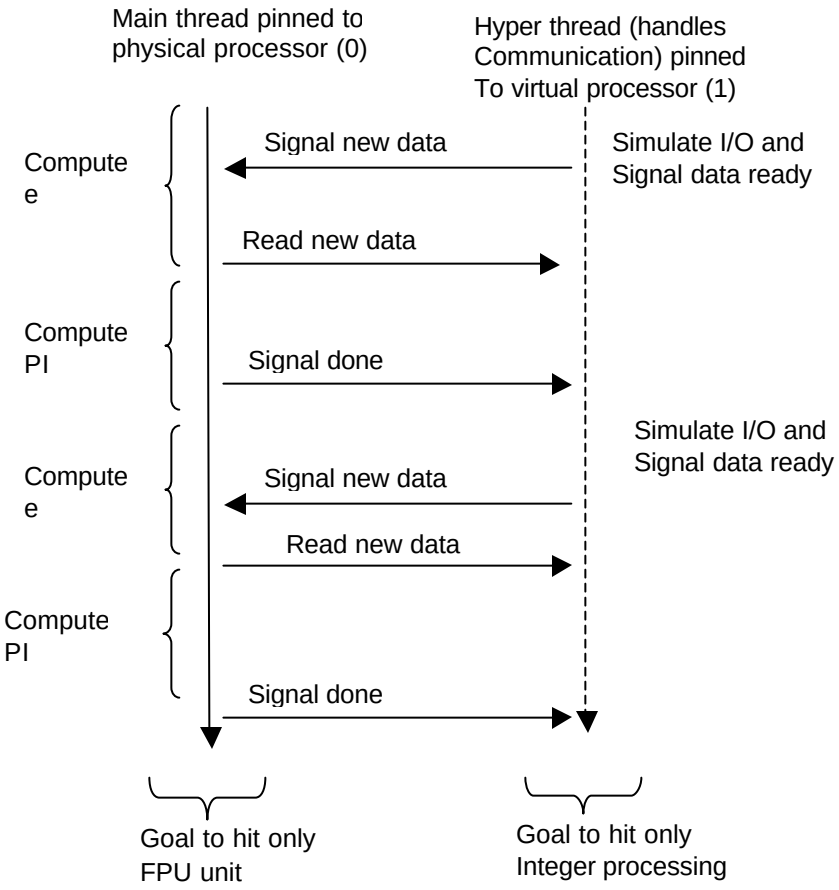
?? Threaded communication / computation on a hyperthread-enabled processor

- A shared memory multi-threaded implementation intended to be the showcase for the hyper-processor capability.

Test Program Setup

Our test program is designed to simulate a typical scientific application, defined as, a processor intensive application with some communication, but optimized in a way that the processor is utilized all the time (non-blocking I/O). The test program consists of 3 parts, a simulated I/O call, computation of *p* (Pi), and another (independent) computation of *e*. The test program designed to take full advantage of the SMT architecture is an application with 2 threads of execution: a thread that simulates I/O operations and a thread that does the computational work. The work thread alternates between calculating the value of *e* and calculating the value of *p* for a given number of iterations. There is a global variable that each thread will busy wait on, if they are waiting on each other, either for a value to be calculated or a communication to complete. We setup the I/O simulation to always last less than or equal time to the calculation of *e*, roughly 15%. To simply things further, the number of iterations to compute *e* and *p* are rigged so that the time they take is roughly equivalent. While *e* finishes calculating, the I/O thread spins on a global variable and executes a noop operation waiting for the computation to complete and another I/O call can be simulated. The I/O thread then sets a global variable that allows *p* to be executed immediately after *e* finishes being calculated, thus allowing for the case where a communications call take longer than the *e* computation. We simulate a network read/write by doing memcpy() operations repeatedly and counting to MAX_LONG and setting a variable equal to each value, thus producing a large number of non-floating point operations that in theory will not conflict with the computational work being scheduled in parallel on the same CPU.

The following is a simple flow diagram for our test application:



Kernel Source Modifications

The goal is to have a pair of processes executing always at the same time and if either is context switched out, then both would stop executing. The purpose is to isolate the processors for testing the hyperthreading of the processors.

Our test kernel is the latest as of this writing, version 2.5.68 and utilizes the O(1) scheduler. Most modifications were made in the schedule() procedure, as highlighted in the following snipit, to force certain threads to be executed on together on the same processor.

sched.c

```
pick_next_task:
        if (unlikely(!rq->nr_running)) {
#if CONFIG_SMP
                load_balance(rq, 1,
cpu_to_node_mask(smp_processor_id()));
                if (rq->nr_running)
                        goto pick_next_task;
#endif
                next = rq->idle;
                rq->expired_timestamp = 0;
                goto switch_tasks;
        }

        array = rq->active;
        if (unlikely(!array->nr_active)) {
                /*
                 * Switch the active and expired arrays.
                 */
                rq->active = rq->expired;
                rq->expired = array;
                array = rq->active;
                rq->expired_timestamp = 0;
        }

        /* danger danger we will crash no doubt */
        /* hyperthread will be pinned to an odd processor
         * currently only support 2
         * modified by nik/frank 4/29/2003
         */
#ifdef CONFIG_X86_HT
        schedCount++; // counter variable  for num sched call

        //if(schedCount%100==0) printk("processor id = %d\n",
smp_processor_id());
#warning "Nikola/Frank HT mod enabled"
        // assume odd processors are hyperprocessors
        if (smp_processor_id() == 1 && myBuddy != NULL
                                && myBuddy->state == 0) {
  // we have to make sure buddy is on hyper-cpu - if so-migrate
                if (task_cpu(myBuddy) != 1) {
                       rq = task_rq_lock(myBuddy, &flags);
                          set_task_cpu(myBuddy, 1);
                          task_rq_unlock(rq, &flags);
                }
                next = myBuddy;
        }
```

```
          else {
                // we are a normal proc and lets do the
right thing
                idx = sched_find_first_bit(array->bitmap);
                queue = array->queue + idx;
                next = list_entry(queue->next, task_t,
run_list);
          }

          // we currently only support 2 procs - need a global
array/struct
          if (smp_processor_id() == 0) {
                if (prev->processorBuddy != NULL && prev !=
next) {
                        myBuddy = NULL;
                // invoke scheduler on hyperthread cpu
                        preempt_disable();
                        smp_send_reschedule(1);
                        preempt_enable();
                }
                else
          if (next->processorBuddy != NULL && myBuddy == NULL)
{
                        myBuddy = next->processorBuddy;

                // invoke scheduler on hyperthread cpu
                        preempt_disable();
                        smp_send_reschedule(1);
                        preempt_enable();
                }
                else {
                ;
                }
          }
#elif
          idx = sched_find_first_bit(array->bitmap);
          queue = array->queue + idx;
          next = list_entry(queue->next, task_t, run_list);
#endif

switch_tasks:
          prefetch(next);
```

Modifying the scheduler entailed adding two elements, a task_t* processorBuddy and a task_t* processorBuddyOwner, that identify to them their corresponding buddy task running on the hyper thread, and a global task_t* myBuddy pointing to the current hyperthreaded task. Our test program calls a custom system call to manipulate these variables and define a task to be put on a hyper-processor. These values are initially null. We decided to introduce a new system call hyperpin() (found in kernel/hyperpin.c) which will set these values to point to a given master thread (buddyOwner) and slave thread (buddy). When the system call is executed it will queue the master on processor 0, by calling set_cpus_allowed(master task, bitmask containing only CPU 0). Then whenever the scheduler (running on CPU 0) sees a task that has a "buddy" it will set the global myBuddy, migrates the "buddy" task to CPU 1 if necessary and signals CPU 1 to execute the scheduler. CPU 1's scheduler will always schedule myBuddy when it is set, if it is not set it will schedule using whatever is next in the run queue. We relay on the load balancer to prevent normal processes running on CPU 1 from being starved. When the master thread is unscheduled then myBuddy is unset. Also when either process exits myBuddy, buddyOwner, and buddy are all reset. The code currently, is meant for demonstration purposes only it does not handle multiple buddy – buddyOwner pairs, nor multiple SMT enabled CPUs. None the less, the system call will make some assertions to protect from improper use which may result in an unstable system.

Results

Average of 3 Runs—all times in seconds – lower is better

Machine Configuration	Serial	Threaded	SPMD
Single Processor w/o SMT*	190 s	271 s	190 s
Single Processor w/ SMT	190 s 0%	164 s ** -35% baseline w/ hyperpin() call	224 s +18% baseline
Dual Processor w/o SMT	189 s 0%	141 s -48% baseline 14% faster than hyperpin	96 s -98% baseline 58% faster than hyperpin

* Baseline Test
** Optimized SMT aware threaded application
Threaded = Threaded application with communications on one thread and computations on another
SPMD = ½ communication + ½ computation running in two separate processes

The center cell represents the time to execute our SMT optimized application which shows a significant improvement over the unoptimized serial application for only a relatively small part amount of communications time (recall: about 15% of 1/2 the combined computation times). Another observation is the alarmingly high penalty incurred for the SPMD application running with SMT enabled, compared with the singe w/o SMT and the dual w/o SMT. The dual processor achieves a very ideal 2x speedup since there is no real communications or synchronization between the two processes. We have concluded from this data that SMT technology can have excessive performance degradation of scientific application if used blindly, but if a hyperthread kernel combined with a hyperthread aware application some significant performance boost can be achieved.

References

1. The IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture (Order Number 245470).
   ftp://download.intel.com/design/Pentium4/manuals/24547011.pdf

2. The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Order Number 245471).
   ftp://download.intel.com/design/Pentium4/manuals/24547111.pdf

3. The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide (Order Number 245472).
   ftp://download.intel.com/design/Pentium4/manuals/24547210.pdf

4. D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *23rd Annual International Symposium on Computer Architecture*, May 1996.
   http://citeseer.nj.nec.com/cache/papers/cs/7286/http:zSzzSzwww.csrd.uiuc.eduzSz~ece412zSzpaperszSztullsen_ISCA96.pdf/tullsen96exploiting.pdf

5. Download of performance libs
   http://www.intel.com/software/products/global/eval.htm#perflib

6. Pentium optimized libraries
   http://www.intel.com/software/products/ipp/ipp30/index.htm

7. Detailed Article on Hyper-threading on the Pentium Xeon
   http://developer.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p01_abstract.htm

8. Intel Processor Programming Manuals
   http://developer.intel.com/design/Pentium4/manuals/

9. Pentium 4 and the G4e: architectural Comparison
   http://arstechnica.com/cpu/01q2/p4andg4e/p4andg4e-6.html

10. IBM Hyperthreading architecture article
    http://www-106.ibm.com/developerworks/linux/library/l-htl/

11. Linux Cross-Reference Site – Linux Source Code hyper-linked browsing
    http://lxr.linux.no/

Installation and Performance Evaluation of ASCI sPPM

Introduction

The sPPM benchmark solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method). The code is written to simultaneously exploit explicit threads for multiprocessing shared memory parallelism and domain decomposition with message passing for distributed parallelism. We focus mainly on MPI and OpenMP for this benchmark. sPPM's primary MPI calls are to MPI_Allreduce, MPI_Isend, MPI_Irecv, and MPI_Wait, therefore mainly asynchronous calls are used and we will run with the file I/O routines disabled (timing only results are thrown away), so we would expect close to 100% CPU utilization for these runs.

Experiment Setup

The sPPM benchmark was downloaded from http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/sppm1.1.tar. A makefile was created for Intel/Linux pointing to the MPI Fortran and C compilers. Since sPPM uses both OpenMP and MPI within the Fortran code we needed to rebuild the MPI libraries with icc compiler because the current libraries were built with the gcc compiler which will not link correctly with the Fortran OpenMP libraries in icc. We built the mpich-1.2.5 libraries and linked to them explicitly in our make file to overcome this problem. The built mpich libraries are located in /home/fjcastan/asci/mpi/mpich-1.2.5. We also had to

explicity link to /usr/lib/gcc-lib/i386-redhat-linux/3.2/libg2c.a to support the “Fortan 2 c” compatibility routines.

The following is the section added to the Makefile :

```
##########################################################################
###### Intel ##############################################################
##########################################################################

SYS= POSIX

#FC = mpif77    #fortran gcc (does not support openmp)
#LD = mpif77    # "
#CC = mpicc     # "

FC = /home/fjcastan/asci/mpi/mpich-1.2.5/bin/mpif77 #        # Fortran compiler
LD = /home/fjcastan/asci/mpi/mpich-1.2.5/bin/mpif77 #        # loader
CC = /home/fjcastan/asci/mpi/mpich-1.2.5/bin/mpicc #        # C compiler
M4 = m4 -Uformat #                  # m4 preprocessor
CPP = gcc -E #                      # cpp preprocessor

#CPPOPT= -DNOMPI -DGNU -DDEBUG  #     # don't use MPI
CPPOPT= -DMPI -DGNU -DDEBUG #        # use MPI
LIBDIR = #                           # MPI library path
INCDIR= #-I/usr/lpp/ppe.poe/include  # MPI include path

#THMODE = -DTHREADED=0  #             # don't use threads
#THMODE = -DTHREADED=1  #            # use direct pthreads calls
THMODE = -DTHREADED=1 -DOPENMP=1 #  # use OpenMP for threads
OMPOPT= -openmp #                    # Fortran OpenMP option
COMPOPT= -openmp #                   # C OpenMP option
THLD= #                              # threaded load options

FPSIZE= -DREAL=float  #            # single precision reals
#FPSIZE= -DREAL=double  #          # double precision reals
TOPT=   #                           # double precision options
FOPT3=   #                          # Fortran compiler options, double precision

MOPT=  -DBOBOUT=0 -DDUMPS=0 -DNOCHDIR=1  # no dumps and no directory change
LIBS= /usr/lib/gcc-lib/i386-redhat-linux/3.2/libg2c.a -lpthread
COPT= -O3 -DF2C=1 -DGNU          # C compiler options
```

1. SYS=POSIX will be used if you enable compilation of POSIX threads instead of OpenMP threads, this specifies to use standard unix POSIX threads if OpenMP is disabled.

2. The –DGNU definition enables some code that had to be added in order to support the icc & gcc limitations.

3. By switching the comment on the CPPOPT you can enable/disable MPI and by swithing the THMODE you can enable OpenMP, posix, or no threads
Needed to link special libraries into the compile for fortan 2 C support since the application uses both C and fortran. (C for I/O and native thread support) currently the makefile supports several builds: MPI w/ OpenMP, MPI w/ pthreads, MPI w/o threads, OpenMP only, and pthreads only. All have been tested and are working.

The –GNU function enables a modification to the Fortran code to not use complex (intrinsic) functions in the parameter definitions, since icc and gcc do not support this. The POSIX pthread code was modified to use constant definitions for several functions which appear to have been added after the original code was written and did not have constants, which was causing compile errors since the C function names have different values depending on the f2C definition of the target platform. The last obstacle was coming up with the correct linker and compiler options for f2C support and so forth, ie. (f2C=1).

The list of modified files are:

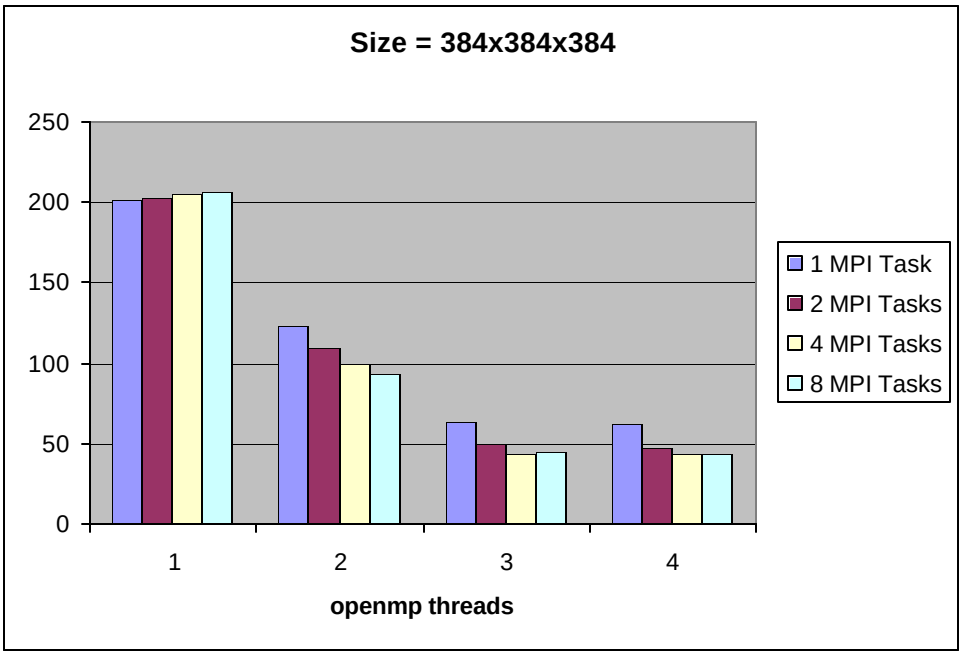
```
Added:    buffers_GNU.h
Modified: cpthreads_sppm_POSIX.c
Modified: main.m4
Modified: Makefile
Modified: runhyd3.m4
Modified: sync.h
```

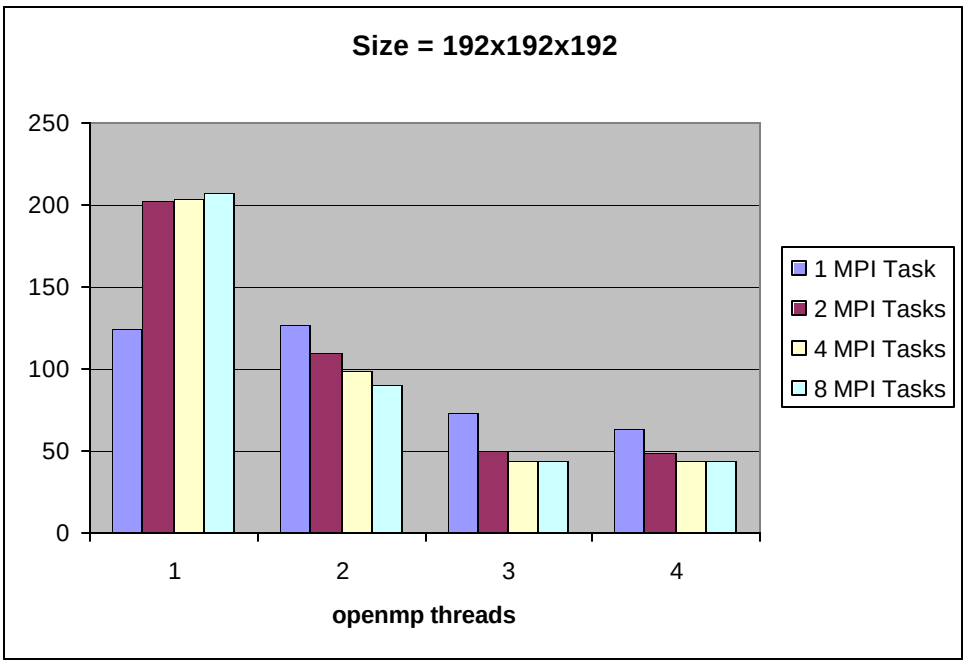
Source, binaries and output files located in /home/fjcastan/asci/sppm

Also note that the application must be run using the mpirun built with icc in /home/fjcastan/asci/mpi/mpich-1.2.5/bin.

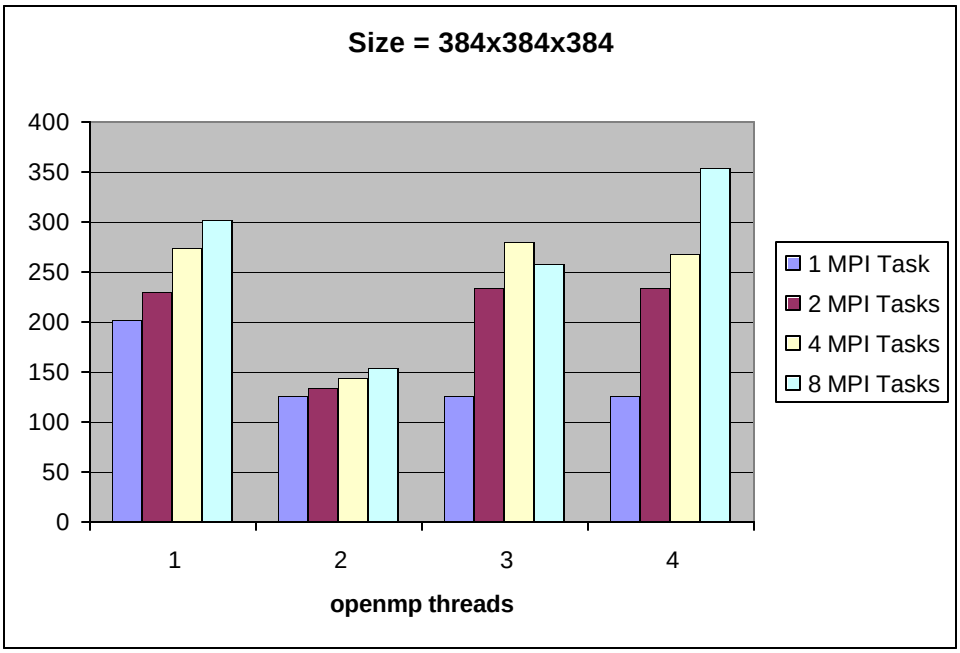
Results

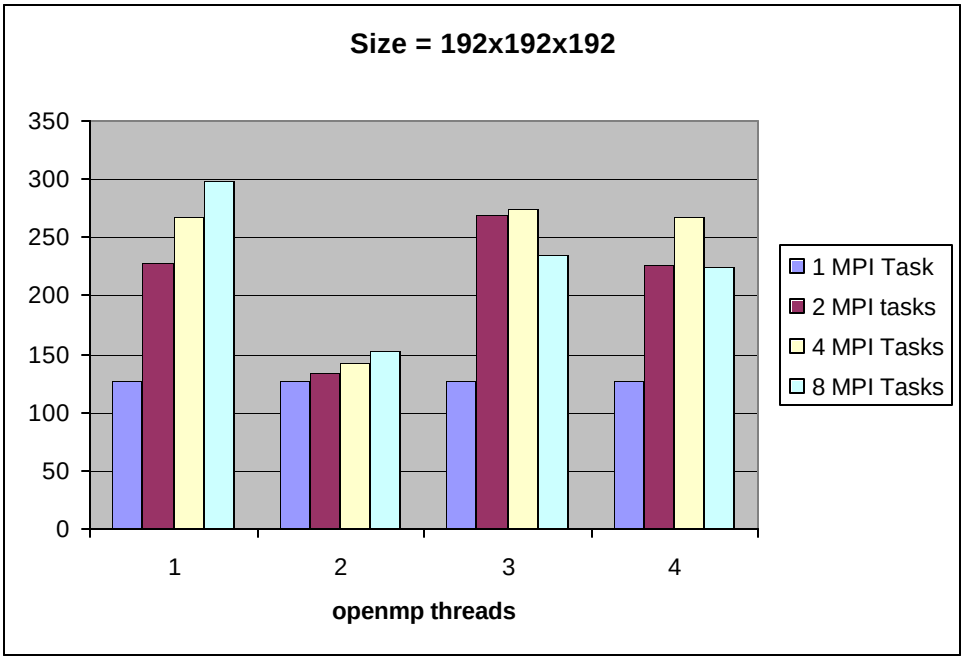
The following are the CPU only times for the sPPM runs on two data sizes:





The following are the overall times for the same runs:





The first two charts show the CPU time only which shows a decent reduction in CPU time per processor as MPI tasks increase, and some decrease in CPU time from 1 to 2 OpenMP threads, and little or no change above 2 threads. This makes sense since there are only 2 physical processors per node for the OpenMP behavior, the MPI behavior also is as expected since the load is distributed among more nodes. The next two charts show total time which includes computation time and communications time. 1 MPI task performs best every time, this seems to indicate a serious communications bottleneck, the next logical step would be to increase the data size to see if better speedups are achieved, unfortunately icc currently has a limitation on the size of arrays in Fortran which prevents us from running any tests with larger data sizes than 384x384x384.