Evaluation of the sPPM Benchmark on the Cray X1

Sarah Anderson
Cray Inc
1340 Mendota Heights Road, Mendota Heights, MN 55120, Phone (651) 605-8945 `saraha@cray.com`

Scott Parker
National Center for Supercomputing Applications
152 Computing Applications Building, 605 East Springfield Avenue, Champaign, Illinois 61820, Phone (217) 244-3310, `scottp@ncsa.uiuc.edu`

Dave Strenski
Cray Inc
7077 Fieldcrest Road, Suite 202, Brighton, Michigan 48116, Phone (313) 317-4438, `stren@cray.com`

ABSTRACT:
The sPPM benchmark code solves a three dimensional gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. It utilizes domain decomposition with message passing for distributed parallelism and may also simultaneously exploit threading for multiprocessing shared memory parallelism. This talk will demonstrate how sPPM was ported and optimized on the Cray X1 and achieved 3.7 Gflops on a single MSP and 112 Gflops on 32 MSPs. We'll discuss the relative performance advantages involved in running sPPM and similar codes in possible combinations of the SSP, MSP, OpenMP and or MPI programming models, and make comparisons of its performance on the Cray X1 with that of the Itanium2 Linux cluster located at NCSA.

KEYWORDS:
Cray X1, sPPM

Introduction

For the past several months Cray has been working with the NCSA applications group on porting and optimizing their benchmarks for the Cray X1. Since the architecture is a multi-streaming vector machine, some codes ported very well and others did not. This paper will cover the details of one NCSA benchmark, namely sPPM, that ported very well to the Cray X1.

The paper will start with a short history about sPPM and its relationship to the hydrodynamics code PPM. Next, since the Cray compiler offers many different programming models, which can be mapped differently to the hardware, the paper will continue with a discussion of these models in detail and explain the benefits of using one model over another. That section will help to illustrate which programming model works best for sPPM. Finally the paper will wrap up with some performance results on the Cray X1, showing how the combination of MPI, streaming, and vectorization yields the best performance. This last section will also compare these Cray X1 results with the Itanium2 Linux cluster located at NCSA.

sPPM Numerics and History

The benchmark sPPM is a simplified version of hydrodynamics code PPM (the Piecewise Parabolic Method) which was developed by Woodward and Colella in the 1980s at Lawrence Livermore National Laboratory. The particular version of PPM that sPPM is derived from solves the nonlinear one dimensional Riemann shock interface problem at each grid cell in Lagrangian coordinates, remapping the solution to the source Cartesian mesh. Strong shocks and discontinuities are preserved by PPM with judicious application of shock steepening, contact discontinuity detection, and the namesake 4th-order accurate interpolation method. Numerical noise is reduced by an addition of an estimated diffusion velocity, essentially a parameterized fluid viscosity. The character of PPM is incorporated in sPPM, but complicated interpolation and diffusion algorithms are replaced with basic equivalents.

The sPPM numeric kernel updates a one dimensional strip of zones, resulting in a high level of data locality and reuse which maximizes the performance of register and cache memory. The kernel is supplied with strips of zones which include boundary or adjacent domain information in "halo" zones. During each strip update, halo zones are gradually "consumed" by the series of decreasing extent loops. For sPPM the boundary halo width is 5 zones on each end of the current sweep direction, and 2 zones on each of the transverse sweep direction. The vectorized sPPM kernel is very regular, requiring 674 floating point operations per grid cell update (counted by the Cray X1 hardware performance counter). The fraction of computational work performed on non-halo zones for a strip of length `n` is approximately `1/(1+5/n)`, which for a domain size as small as 16 results in a "real update" efficiency of 76%. A more typical strip size of 128 zones is over 96% efficient in this regard.

The code is generalized in three dimensions by directional splitting, applying the one dimensional kernel in each mesh direction in succession. Shared memory OpenMP can be used to update the independent blocks or "pencils" of the one dimension strips. This organization again localizes memory references and provides an efficient method of transposing to and from strip order. Throughout the code, memory copies are done infrequently. Extending parallelism further, sPPM can use a rectangular domain decomposition in which participating processors exchange updated boundary information between directional sweeps. This communication is done with simple MPI message passing. This makes good use of high bandwidth interconnects and is relatively latency tolerant because there are so few messages. Since sPPM was conceived as a simple benchmark code, it does not attempt to overlap communication, computation nor I/O. The code also makes no attempt to load-balance across processors, although the more modern implementations of PPM does all of these.

Programming Models

The Cray X1 programming environment is versatile because it contains so many different programming models. The programmers can use vector processing and multi-streaming at the lowest level, along with the application program interfaces (API) for: multi-threaded, shared memory parallelism (OpenMP), message passing interface (MPI), shared memory access library (SHMEM), Co-Array FORTRAN (CAF), and Unified Parallel C (UPC). This rich programming environment is layered on top of a mix of hardware, single streaming vector processors (SSP), multi streaming vector processors (MSP), tightly coupled shared memory nodes, all connected in a low latency, high bandwidth hypercube.

To illustrate these programming models, consider the following four nested loop structures. The loop marks to the left of the code are similar to those outputted in the listing file by the compiler when the `-rm` option is used. The letter `P` indicates that the loop was parallelized, usually by supplying the OpenMP directive `C$OMP PARALLEL DO`, and compiling with the `-O task1` compiler option. The letter `M` indicates that the loop was streamed either automatically by the compiler or with the help of the Cray Streaming Directive (CSD), `!csd$ PARALLEL DO`. Lastly the letter `V` indicates the loop was vectorized, again either automatically or with the help of a compiler directive. If a loop is not parallelized, streamed, nor vectorized the compiler puts the numerical number associated with the level of nesting of that loop, 1 being the innermost loop and counts outward from there.

```
   Example 1  (MSP/OMP)                  Example 2  (MSP/OMP)
P----<   do k=1, num_k               P----<   do k=1, num_k
PM-----<   do j=1, num_j             P2-----<   do j=1, num_j
PMV------<   do i=1, num_i           P2MV-----<   do i=1, num_i
PMV              ... work ...        P2MV             ... work ...
PMV------>   end do                  P2MV----->   end do
PM----->   end do                    P2----->   end do
P---->    end do                     P---->    end do


   Example 3  (SSP/OMP)                  Example 4  (MSP)
P----<   do k=1, num_k               M----<   do k=1, num_k
P2-----<   do j=1, num_j             M2-----<   do j=1, num_j
P2V------<   do i=1, num_i           M2V------<   do i=1, num_i
P2V              ... work ...        M2V              ... work ...
P2V------>   end do                  M2V------>   end do
P2----->   end do                    M2----->   end do
P---->    end do                     M---->    end do
```

Obviously the performance of the different programming models will depend greatly on the limits of the loops, num_i, num_j, num_k. The first example above has each loop distributed with a combination of vectroization, streaming, and threading. The innermost loop is vectorized on the pipes within the SSP, the middle loop is streamed within the MSP across the four SSPs, and the outer most loop is threaded across the four MSPs with in the shared memory node. As long as the inner loop has a trip count greater then the vector length (64), this code should perform well. There are only four SSPs per MSP and only four MSPs per node, so these trip counts only need to be greater then four for good performance. For good performance the programmer also needs to consider the overhead associated with parallelization. Since streaming occurs at the lowest level (instruction level) it has the fastest context switching time and lowest parallel overhead as compared to threading with OpenMP, so its favorable to stream instead of threading from an overhead point of view.

Example 2 illustrates what might happen if the innermost loop is contained within a subroutine by itself. By default the compiler will not look at the calling subroutine and will only have the option of streaming and vectorizing this single loop. This is fine if the loop i has a very long trip count. Again the loop is threaded on the outer most loop using OpenMP, and the second loop is simply carried along in the parallization.

Example 3 illustrates an interesting programming model in that the streaming level has been removed altogether. In this model the vectors are directly connected to the threading layer, by running the OpenMP across sixteen SSPs per node. This is accomplished by using the -O spp compiler directive. This can be an effective programming model since vectorization has the inner loop all to itself and the sixteen threads are all managed by OpenMP to saturate the node.

Example 4 is based on the standard MSP model. The compiler finds some high level loop and streams across it. Vectorization is used on the lowest level. The limitation of this loop is that it can only scale to one MSP or 1/4 of the node, but has a lower overhead than OpenMP since the streaming is done on an instruction level.

Programming Models Applied to sPPM

With an understanding of these programming models, let us now explore how to best map the sPPM benchmark to the Cray X1 node. When the code was first ported to the Cray X1, the performance was about two Gflops per MSP. This was due to the structure of the code. Here is a high level view of the loop structure.

```
do n=1,num_domains                MPI domain Layer
  do mypen=1,nypens * nzpens      OpenMP pencil Layer
    do k= z_start, z_stop     \
      do j = y_start, y_stop   \
        do i = x_start, x_stop \ Extract pencil
        end do                 /   of y/z data
      end do                  /
    end do                   /
    do izy=1,(y_stop-y_start)*(z_stop-z_start)
      do i = 2-nbdy, n+nbdy-1       \
      end do                         \
      :                               \
      : Several single nested loops    } sppm routine
      :                               /
      do i = 2-nbdy, n+nbdy-1        /
      end do                        /
    end do                    End strips within a pencil
    do k= z_start, z_stop     \
      do j = y_start, y_stop   \
        do i = x_start, x_stop \
        end do                 / Store pencil data
      end do                  /
    end do                   /
  end do                      End OpenMP pencil layer
end do                       End MPI domain layer
```

The outer most leaf of the subroutines is the sppm subroutine. This routine is composed of a series of singly nested loops that have a trip count on the order of the `x` dimension of the sub-domain. Since these are single nested loops, the compiler has to stream and vectorize this trip count. Two problem sizes were examined, one with a sub-domain grid of 192 x 192 x 192 in the x, y, z and the other with a grid size of 384 x 384 x 384. For the initial problem size of 192 x 192 x 192 the machine had to divided 192 by four, one for each stream, and give each SSP a trip count of 48 elements, small relative to the machine's vector length of 64 elements. For the 384x384x384 problem, each SSP gets a trip count of 96 elements - better but still not long enough for great performance. To get the optimal performance, the compiler needs to pass the full leading dimension to the vector pipes on the SPP. To accomplish this, we need to compile the `sppm` subroutine in SSP mode with the compiler option `-O ssp`. By compiling a single streaming version of the subroutine, the compiler only vectorizes these single nested loops. On the calling side the compiler directive, `!csd$ parallel do`, is used to tell the compiler to stream across the calls to `sppm`. This is what that code sample looks like.

```
     !dir$ ssp_private sppm
     !csd$ parallel do
           do 1200 izy= 0, (izstop-izstart+1)*(iystop-iystart+1)-1
             iz= izy/(iystop-iystart+1) + izstart
             iy= mod( izy, (iystop-iystart)+1 ) + iystart
             call sppm( xl, rrho, pp, uux, uuy, uuz, iy+noffy, iz+noffz,
          &               rhonu(1-5,iy+noffy,iz+noffz),
          &               pnu  (1-5,iy+noffy,iz+noffz),
          &               uxnu (1-5,iy+noffy,iz+noffz),
          &               uynu (1-5,iy+noffy,iz+noffz),
          &               uznu (1-5,iy+noffy,iz+noffz),
          &              dtime, cournt, gamma,
          &              smlrho, smalle, smallp, smallu, nx, 5 )
      1200 continue
     !csd$ end parallel do
```

Note the use of the CSD directive, `!dir$ ssp_private sppm`. This is needed to tell the compiler that a single streaming version of `sppm` exists and it is safe to multi-stream the loop. This optimizes the lowest two loop levels, vectorizing the many single nested loops within the subroutine `sppm` and streaming across the `do 1200` loop. Moving to the next level of loops higher, the `do mypen=1,nypens*nzpens` can be threaded if needed. On one hand threading this loop with OpenMP would be more efficient since the communication between processors within OpenMP is cheaper than moving data between processors using MPI. In addition, coarse grain parallelism (few large tasks vs. many small tasks) is usually more efficient.

Both of these would imply that threading at the `pencil` loop would be more efficient. However in this case it is not. One reason is that the blocks of work do not change greatly by threading at this level. It simply groups four 384x384x384 domains together. The main reason for the lack of additional speed in threading this loop is the resources available for moving data during the MPI communication stage. If the `pencil` loop is threading, the MPI communication will be performed by only one MSP, if the loop is not threaded, each block is under its own MPI processes, so we have four MSPs involved in the MPI communication.

Performance of sPPM on the Cray X1 and Itanium2 Linux Cluster

Before talking about the results, the problem size has to be considered since the performance can be greatly dependent on size. In order to give each programming model an equal chance of performing well, a problem size of 384x384x384 per MSP was chosen. This ensures that each MSP has enough work, and that the lowest level vector work has long enough trip counts to perform well. Another decision was to measure performance on a per node basis instead of a per MSP or SSP basis. Since we are mixing MSP and SSP programming modes with both OpenMP and MPI programming, the least common multiple is the node. Recall that the Cray X1 node can be four MSP processors running OpenMP or MPI, or sixteen SSP processors running OpenMP or MPI. Another reason for benchmarking on multiples of nodes is to minimize variations in the timings. By using all four MSPs on a node, the program has complete control over the memory bandwidth within the node. Even when the benchmark spans multiple nodes, within each node the program has complete control over the nodes memory bandwidth, and is only affected by other jobs using the bandwidth between nodes[1].

To scale the problem past the node, the program needs to use MPI. For the results given below, the 384 x 384 x 384 per MSP problem size was continued for each new node. Therefore each node has a problem size of 2(384) x 2(384) x 384 or 768 x 768 x 384 grid points. From an SSP perspective the problem size is 4(192) x 4(192) x 384 or again 768 x 768 x 384 grid points. Scaling to multiple nodes involves only changing the `Z` dimension. Here is a table of the size configurations for the test cases used with the four different programming models. Note that in all cases the total size is the same on a per node basis.

Nodes	MSP	Model	Threads	MPI			Sub-domain Size			Total Size		
				X	Y	Z	X	Y	Z	X	Y	Z
1	4	MSP_MPI	1	2	2	1	384	384	384	768	768	384
2	8	MSP_MPI	1	2	2	2	384	384	384	768	768	768

4	16	MSP_MPI	1	2	2	4	384	384	384	768	768 1536
8	32	MSP_MPI	1	2	2	8	384	384	384	768	768 3072
16	64	MSP_MPI	1	2	2	16	384	384	384	768	768 6144
1	4	MSP_OMP	4	1	1	1	768	768	384	768	768 384
2	8	MSP_OMP	4	1	1	2	768	768	384	768	768 768
4	16	MSP_OMP	4	1	1	4	768	768	384	768	768 1536
8	32	MSP_OMP	4	1	1	8	768	768	384	768	768 3072
16	64	MSP_OMP	4	1	1	16	768	768	384	768	768 6144
1	4	SSP_MPI	1	4	4	1	192	192	384	768	768 384
2	8	SSP_MPI	1	4	4	2	192	192	384	768	768 768
4	16	SSP_MPI	1	4	4	4	192	192	384	768	768 1536
8	32	SSP_MPI	1	4	4	8	192	192	384	768	768 3072
16	64	SSP_MPI	1	4	4	16	192	192	384	768	768 6144
1	4	SSP_OMP	16	1	1	1	768	768	384	768	768 384
2	8	SSP_OMP	16	1	1	2	768	768	384	768	768 768
4	16	SSP_OMP	16	1	1	4	768	768	384	768	768 1536
8	32	SSP_OMP	16	1	1	8	768	768	384	768	768 3072
16	64	SSP_OMP	16	1	1	16	768	768	384	768	768 6144

Results were collected for five scenarios, the four pure programming models; MSP_MPI, MSP_OMP, SSP_MPI, and SSP_OMP and the best performing version that uses the Cray streaming directives, MSP_CSD_MPI, to stream across the calls to the `sppm` subroutine. The results all used the 384x384x384 per MSP data set size. The values in the table are Gflops with the percent of peak given in (). The peak performance is 12.8 Gflops per MSP or 51.2 Gflops per node.

Cray X1 GFLOPS (%peak)

Nodes	MSPs	MSP_CSD_MPI	MSP_MPI	MSP_OMP	SSP_MPI	SSP_OMP
1	4	14.61 (29%)	9.52 (19%)	11.06 (22%)	10.45 (20%)	13.52 (26%)
2	8	29.57 (29%)	21.49 (21%)	22.20 (22%)	20.88 (20%)	26.16 (26%)
4	16	57.24 (29%)	42.93 (21%)	44.28 (22%)	41.72 (20%)	53.45 (26%)
8	32	118.11 (29%)	85.93 (21%)	88.62 (22%)	83.38 (20%)	104.39 (25%)
16	64	237.08 (29%)	172.39 (21%)	177.56 (22%)	225.79[2] (27%)	208.99 (26%)
31	124	459.61 (29%)	334.77 (21%)	344.00 (22%)	322.97 (20%)	402.19 (25%)

Looking at the results, clearly the MSP_CSD_MPI version is the best performing, sustaining 29% of the peak of the hardware. The two pure MSP models, MSP_MPI and MSP_OMP, do poorly. The main reason here is the innermost loops within the `sppm` subroutine needs to be both streamed and vectorized giving the vector pipe only enough work for one and a half vector slices (384/4=96, 96/64=1.5). A larger problem would help, but the code would run into memory limits soon. The most interesting results are those for the SSP_OMP version that almost matches the MSP_CSD_MPI version. The vectors on the innermost loops are performing efficiently with six vector slices to work with (384/64=6). In addition, the lower overhead of the OpenMP, vs. MPI, is giving it a boost over the SSP_MPI version. In the end, the MSP_CSD_MPI version is running better because the CSD streaming has a lower overhead then the OpenMP in the SSP_OMP version.

For comparison purposes runs of the sPPM benchmark were also performed on an Itanium2 Linux cluster located at NCSA. The cluster consists of 256 dual processor Itanium2 nodes running at a processor speed of 1.3 GHz and having a peak performance of 5.2 Gflops per processor. The nodes are connected with a Myrinet 2000 interconnect. Below are the results for runs on up to 32 nodes, using one processor per node. More information can be found at the NCSA web site, http://www.ncsa.uiuc.edu/UserInfo/Perf/NCSAbench/sPPM.html.

Nodes	CPUs	Threads	MPI			Sub-domain Size			Total Size			GFlops	%Peak
			X	Y	Z	X	Y	Z	X	Y	Z		
1	1	1	1	1	1	384	384	384	384	384	384	0.927[3]	18%
2	2	1	2	1	1	384	384	384	768	384	384	1.843[3]	18%
4	4	1	2	2	1	384	384	384	768	768	384	3.603[3]	17%
8	8	1	2	2	2	384	384	384	768	768	768	7.326[3]	18%
16	16	1	4	2	2	384	384	384	1536	768	768	14.541[3]	17%
32	32	1	4	4	2	384	384	384	1536	1536	768	29.113[3]	17%

As can be seen on both the Itanium2 and Cray X1, the sPPM benchmark can be well tuned for both microprocessor/cache based machines and multi-stream vector hardware. The Cray X1 has an advantage in that it has a higher peak value for the MSP processor and is also able to get a much higher percentage of that peak.

All the results presented in this paper are for 64-bit words. The only exception follows. During the porting and optimization of sPPM, a 32-bit version of the code was compiled and tested on the Cray X1. The results show a higher flop rate, but the percent of peak is lower and based on a 32-bit peak rate of 25.6 Gflops per MSP.

```
Single MSP 64-bit: 3.698 Gflops (29%)
Single MSP 32-bit: 5.722 Gflops (22%)

120 MSPs 32-bit:   663.0 Gflops (22%)      512x 500x 250 problem size
252 MSPs 32-bit:  1396.6 Gflops (22%)     1536x1536x1792 problem size
```

Footnotes

[1] All jobs run on the Cray X1 used the `-N {num}` option on the `aprun` command line. Where {num} is 4 for MSP jobs and 16 for SSP jobs. This forces the job not to span multiple nodes unless needed.

[2]This value looks suspicious, and has not been verified due to a limit on machine resources.

[3]The Itanium2 Linux Cluster Gflops reported in this paper are actually 1/2 of those reported by the hardware performance monitor on the Itanium2. The reason is that the reported values are twice the value of what was expected when compared to the elapsed time used during the program and as compared to other microprocessor hardware platforms. It is felt that either the `MADD` instruction is throwing the flop count off, or the fact that the machine is a duel processor SMP node, has affected the flop count.

References

[1] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R. Woodward, L.J. Shieh, S.W. White,**"Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System"**, LLNL Report UCRL-JC-134237, http://citeseer.ist.psu.edu/mirin99very.html