Cray X1 Optimization:
A Customer's Perspective

Patrick H. Worley

Oak Ridge National Laboratory

46th Cray User Group Conference
May 17, 2003
Knoxville Marriott
Knoxville, TN

Acknowledgements

- Research sponsored by the Atmospheric and Climate Research Division and the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.
- These slides have been authored by a contractor of the U.S. Government under contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes
- Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the United States Department of Energy under Contract No. DE-AC05-00OR22725.

Who Am I?

Senior Research Staff at Oak Ridge National Laboratory, working on

- Performance evaluation of HPC architectures
- Performance analysis and modeling research
- Parallel algorithm design and optimization for global atmospheric and ocean models

and focusing (most recently) on the

- Cray X1
- SGI Altix
- IBM p690 cluster with Federation interconnect

at ORNL.

Phoenix

Cray X1 with 64 SMP nodes

- 4 Multi-Streaming Processors (MSP) per node
- 4 Single Streaming Processors (SSP) per MSP
- Two 32-stage 64-bit wide vector units running at 800 MHz and one 2-way superscalar unit running at 400 MHz per SSP
- 2 MB Ecache per MSP
- 16 GB of memory per node

for a total of 256 processors (MSPs), 1024 GB of memory , and 3200 GF/s peak performance.

General Approach

- Educate yourself about …
  - the architecture and system software
  - the algorithmic characteristics of your code
- Use performance diagnostic tools
  - Loopmarks
  - CPAT performance analysis tool
  - Source code instrumentation (user-inserted timers, …)
  - Third party tools: Tau, mpiP, …
- Iterate
  - Identify performance bottlenecks
  - Adjust compiler optimization, add compiler directives, restructure code, …
  - Run experiments

Basics

- Serial optimization comes first.
  - Target appropriate granularity
  - Determine whether load imbalance causes serial performance problems to appear as communication overhead
- Good performance on the X1 requires good vectorization.
  - Need large vector fraction
  - Need reasonable vector lengths for loops that do vectorize (target is either 64 or 256)
- Memory access patterns can be important.
  - X1 has a cache

However, memory subsystem performance is a strength of this architecture.

Basics (cont.)

- Interprocessor communication performance *can be* excellent.
  - Point-to-point and cross-sectional bandwidth are very high
  - Latency is low if use SHMEM or Co-Array Fortran (or UPC?)
- Multilevel parallelism means that there are multiple opportunities for performance optimization.
  - Vectorization
  - Streaming
  - Threads (e.g. OpenMP)
  - Processes (distributed memory parallelism)
- Multiple interprocessor communication options also provide an opportunity for performance optimization.
  - MPI
  - SHMEM
  - Co-Array Fortran / UPC

Talk Outline

Case Studies* illustrating

- System characteristics
- Application performance diagnosis

(using lots of data to convey a little information).

- For details on optimization tools and techniques, see Cray documentation.
- For additional performance data and case studies, see
  http://www.csm.ornl.gov/evaluation/PHOENIX
  http://www.csm.ornl.gov/~dunigan/cray

* Unless otherwise indicated, data collected by Pat Worley or Tom Dunigan, both of ORNL

Measurement Gotchas

- Task Migration
  - Timings can be (significantly) perturbed if your job is migrated during a timing run.
- Timer overhead
  - Do not use `gettimeofday` for timing. Instead, use `rtc` (Fortran) or `_rtc` (C), to minimize overhead. MPI_WTIME cost is reasonable for coarse grain measurements.
  - Use timers sparingly, and outside inner loops. Even with `rtc`, perturbation due to instrumentation can be high.
  - Run with timers both enabled and disabled, to determine perturbation.
- Performance variability
  - Activity on the front-end and on filesystems have perturbed performance in the past (may be resolved now?)
  - Always measure performance multiple times.

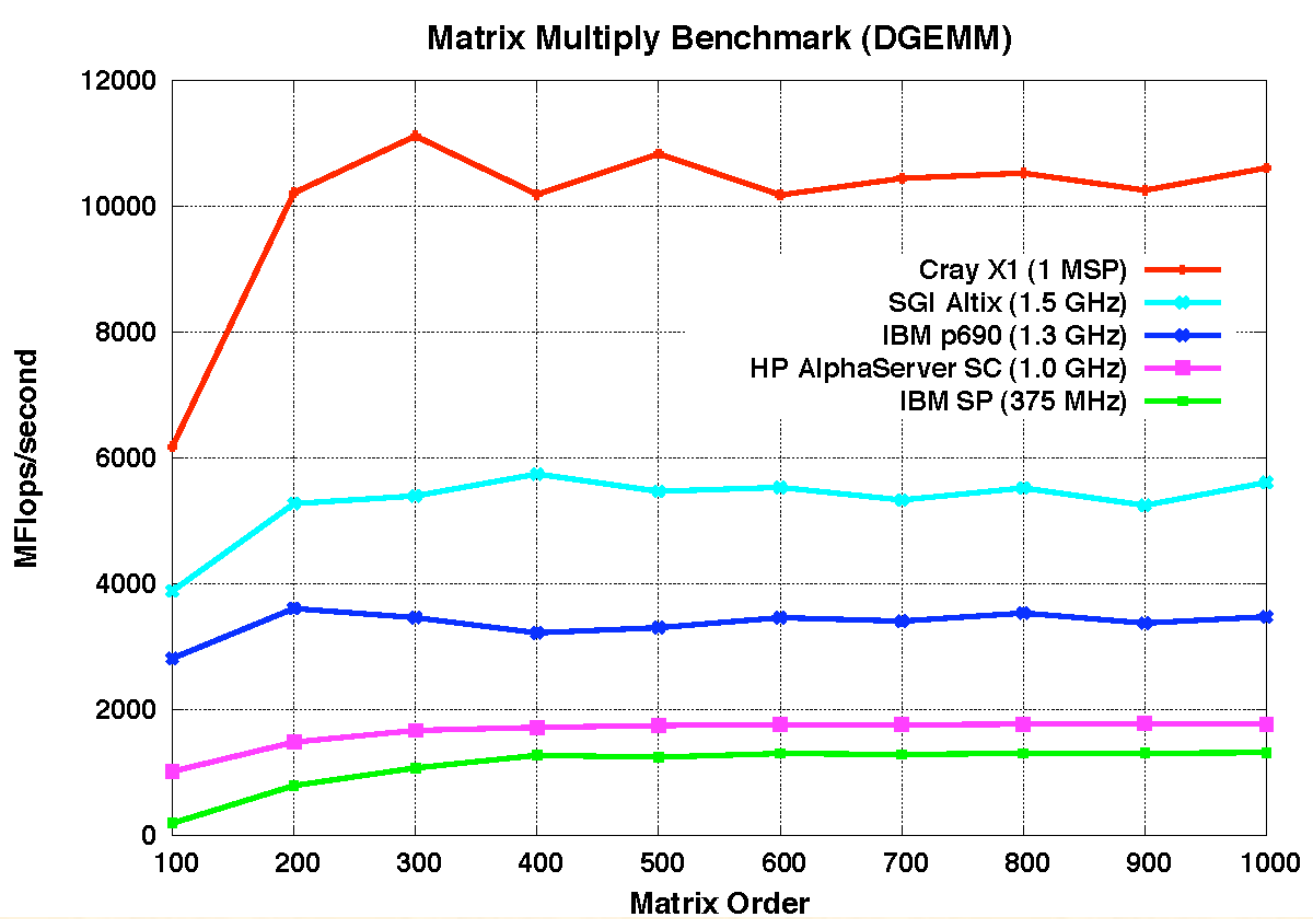
System Characteristics

- Serial performance
  - Performance range
  - Runtime vs. compile time loop bounds
  - Impact of memory subsystem on performance
- Communication performance
  - MPI protocols
  - MPI vs. SHMEM vs. Co-Array Fortran
  - Distance
  - Contention

DGEMM Benchmark

Comparing performance of vendor-supplied routines for matrix multiply. Cray X1 experiments used routines from the Cray scientific library libsci.

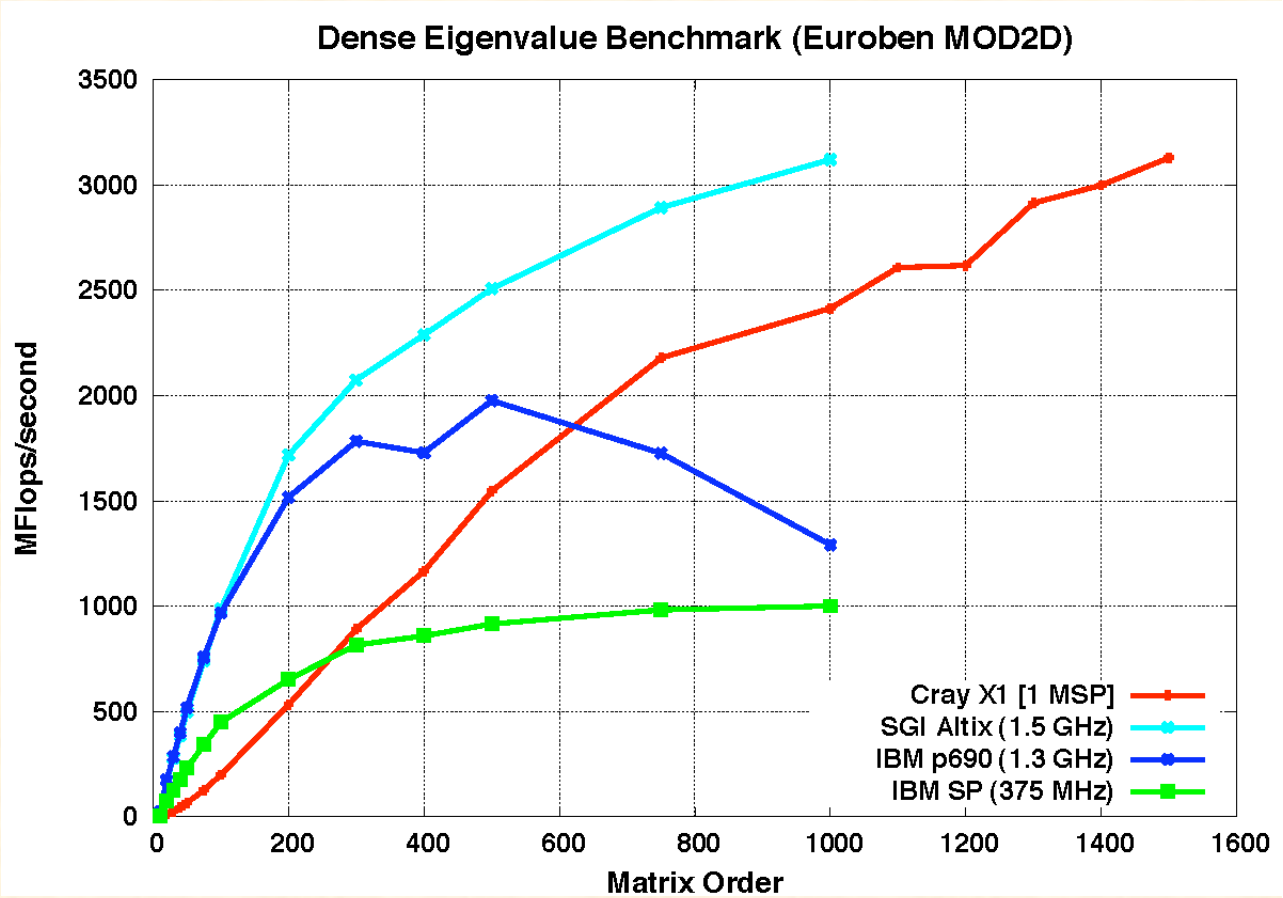
Good performance achieved, reaching 80% of peak relatively quickly.



Euroben MOD2D Benchmark

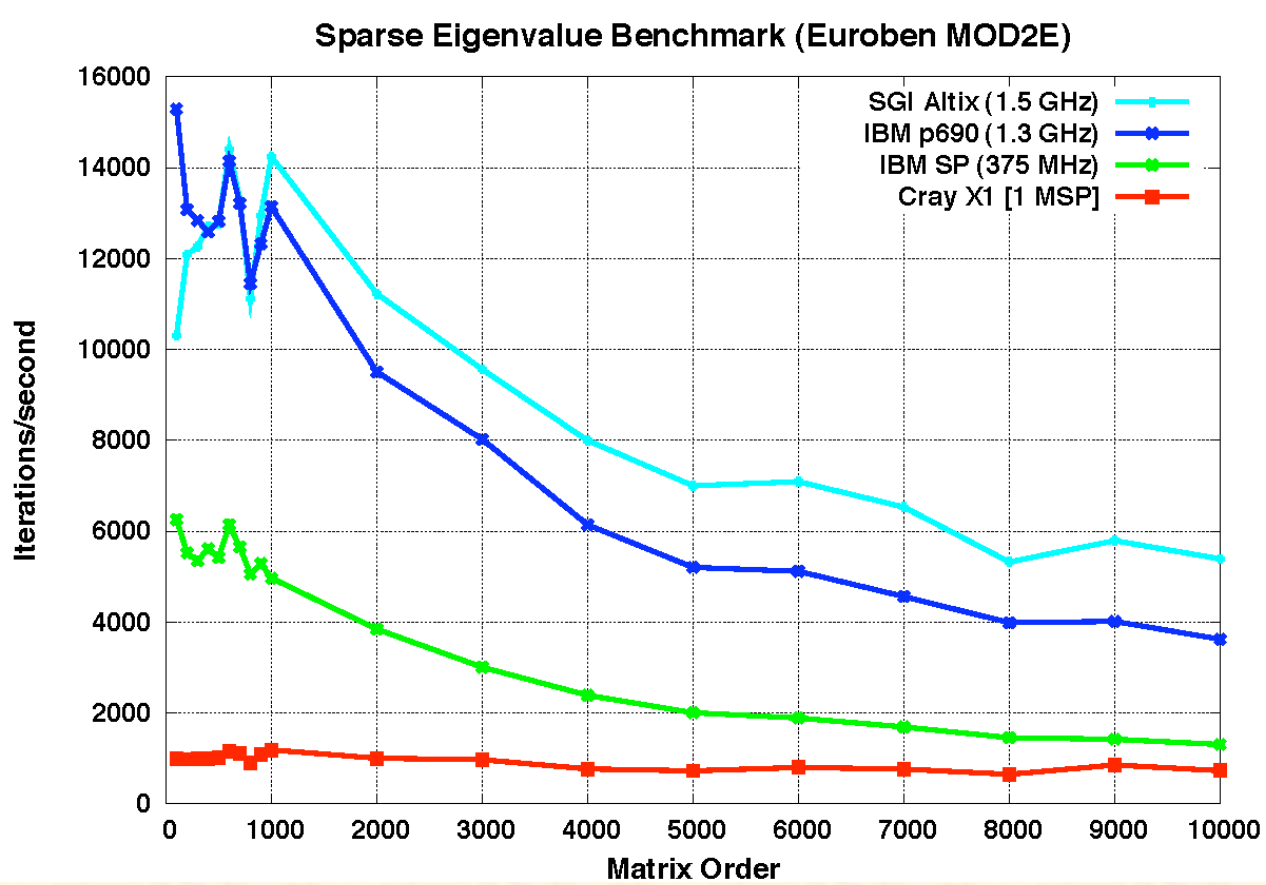
Comparing performance of vendor-supplied routines for dense eigenvalue analysis. Cray X1 experiments used routines from the Cray scientific library libsci.

Performance still growing with problem size for Cray and SGI. Performance of IBM systems has peaked.



Euroben MOD2E Benchmark

Comparing performance of Fortran code for sparse eigenvalue analysis. Aggressive compiler options were used on the X1, but code was not restructured and compiler directives were not inserted. Performance is improving for larger problem sizes, so some streaming or vectorization is being exploited. Performance is poor compared to other systems.



PSTSWM

The Parallel Spectral Transform Shallow Water Model represents an important computational kernel in spectral global atmospheric models. As 99% of the floating-point operations are multiply or add, it runs well on systems optimized for these operations. PSTSWM exhibits little reuse of operands as it sweeps through the field arrays; thus it exercises the memory subsystem as the problem size is scaled and can be used to evaluate the impact of memory contention in SMP nodes. PSTWM is also a parallel algorithm testbed, and all array sizes and loop bounds are determined at runtime. This makes it difficult for the X1 compiler to identify which loops to vectorize or stream.

PSTSWM Experiment Particulars

These experiments examine serial performance, both using one processor and running the serial benchmark on multiple processors simultaneously. Performance is measured for a range of horizontal problems resolutions (T5 - T680) and vertical levels (1 - 66)

Horizontal Resolutions

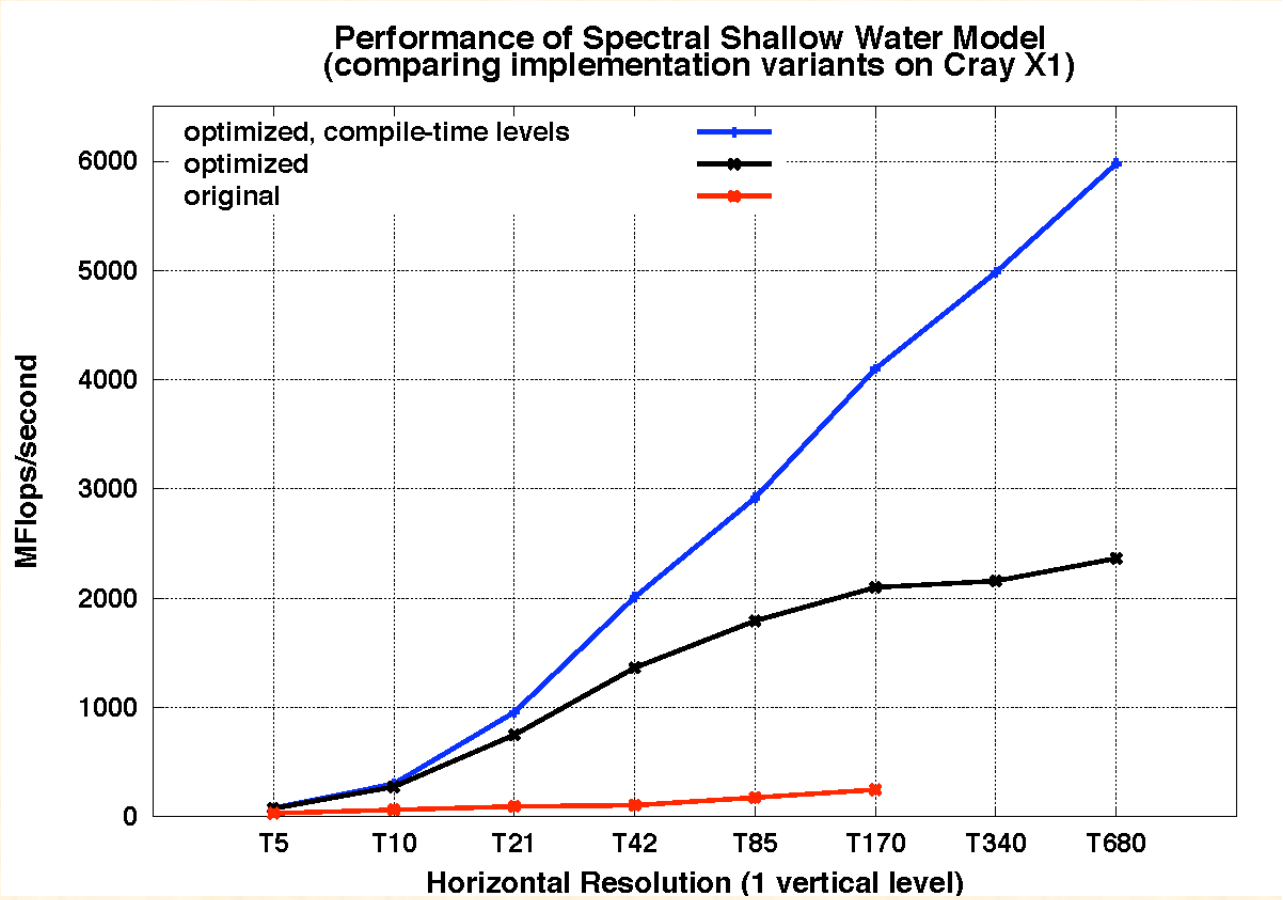
T5:	8 x 16
T10:	16 x 32
T21:	32 x 64
T42:	64 x 128
T85:	128 x 256
T170:	256 x 512
T340:	512 x 1024
T680:	1024 x 2048

PSTSWM Code Versions

- Original (unvectorized) code
- Port to X1
  - changing loops and local array definitions for select routines
- Port to X1 with compile-time specification of number of vertical levels

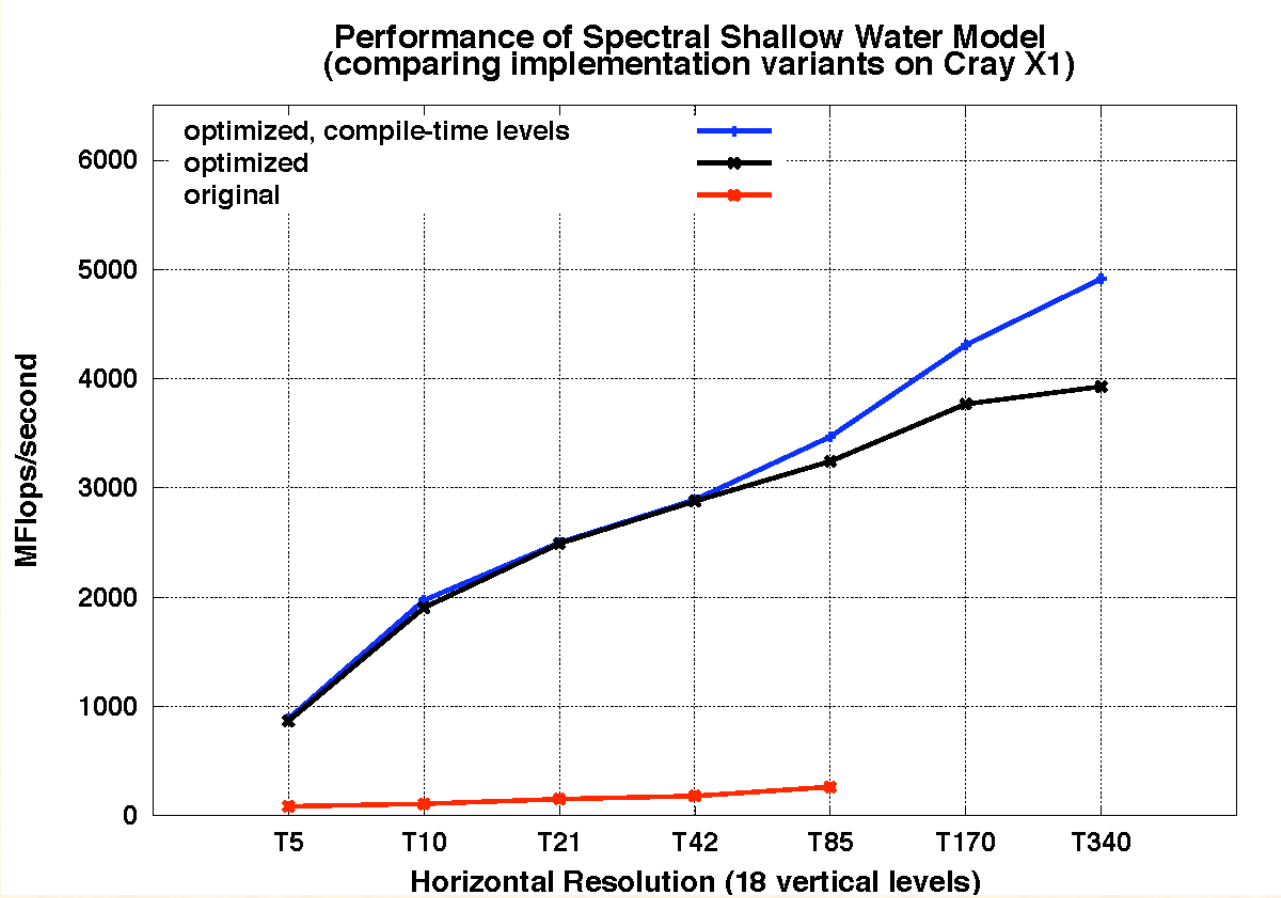
PSTSWM Implementation Comparisons

Comparing performance of different code versions for one vertical level. Code modifications are crucial for this code. Fixing vertical dimension at compile time improves performance for large problem sizes.



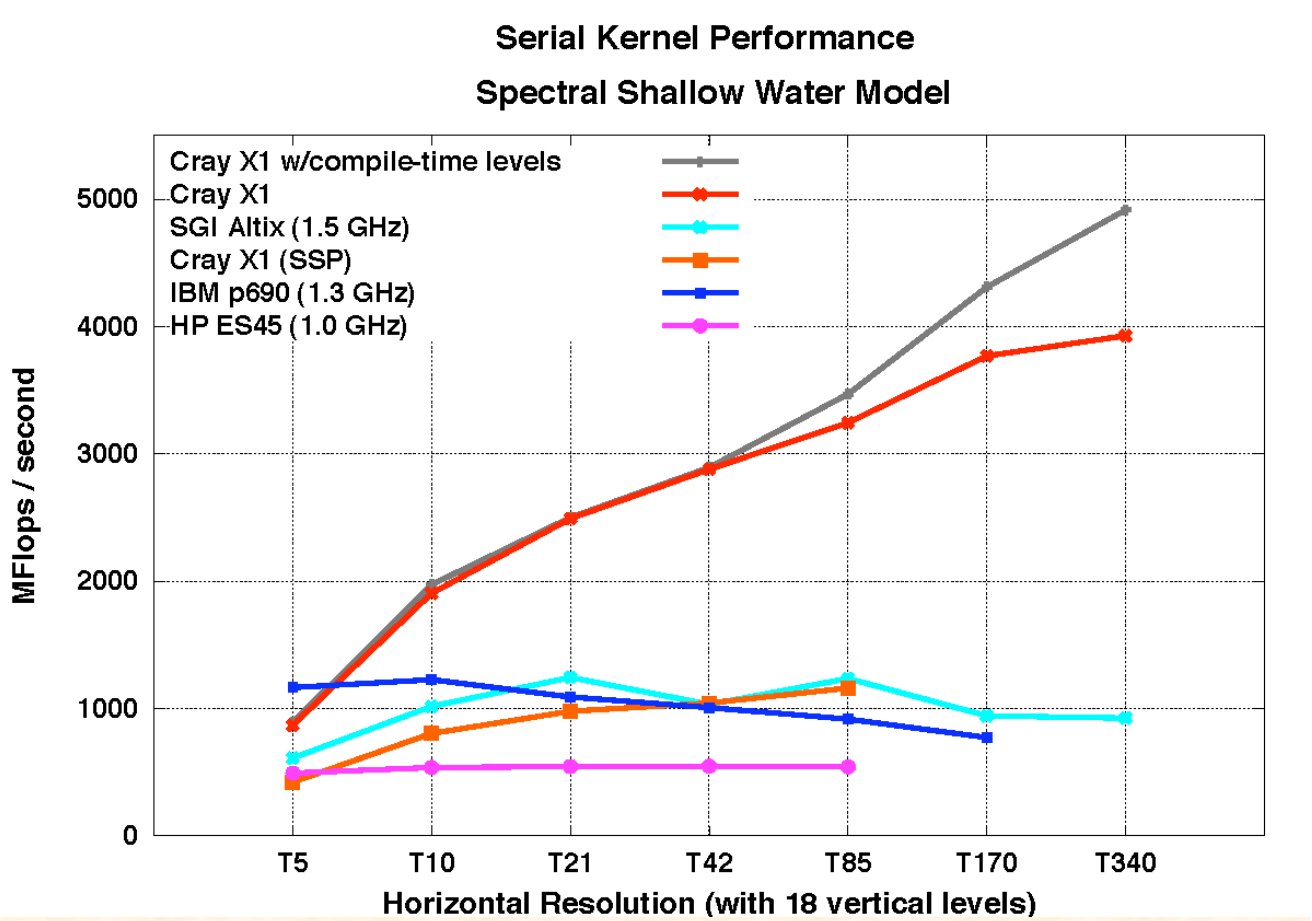
PSTSWM Implementation Comparisons

Comparing performance of different code versions for 18 vertical levels. Improvement due to fixing vertical dimension at compile time not as dramatic as for one vertical level.



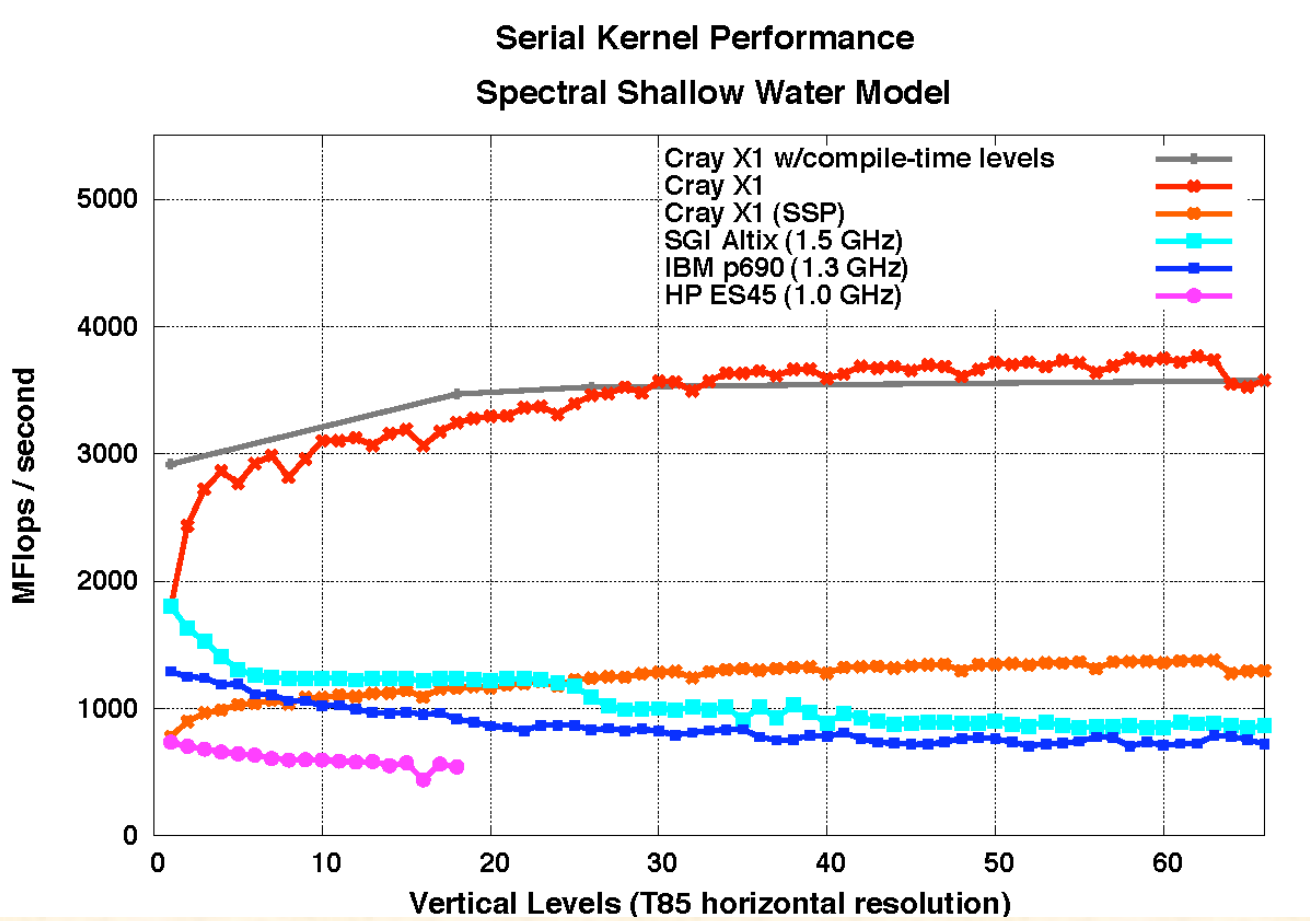
PSTSWM Serial Benchmark

Comparing single processor performance with PSTSWM for 18 vertical levels. X1 MSP version performance scaling well with problem size, and even performance of SSP version exceeds p690 processor performance for the larger problem sizes.



PSTSWM Serial Benchmark

Comparing single processor performance with PSTSWM for T85 horizontal resolution and a range of numbers of vertical levels. Performance of SSP version exceeds that of p690 and Altix processor performance for larger numbers of vertical levels due to the superior processor/memory bandwidth of the X1 .

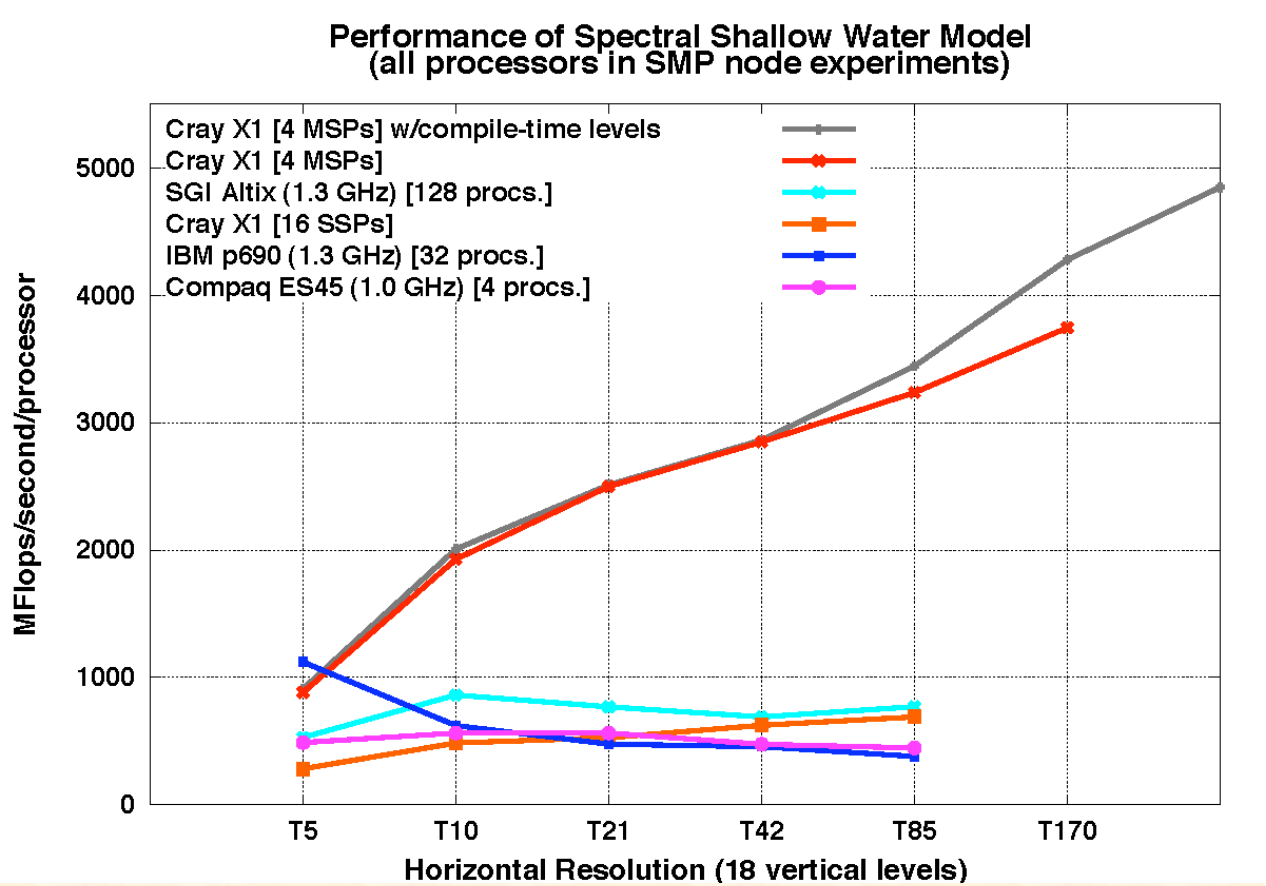


PSTSWM Scaling

- Vertical dimension is a vectorization direction for many phases of the code, so computational rate increases despite greater demands on the memory subsystem.
- PSTSWM has two primary computational phases, a block FFT and a Legendre transform (LT). The FFT (complexity O(N*N*logN)) achieves 25% of peak. The LT (complexity O(N*N*N)) achieves 50% of peak. The larger the horizontal resolution, the more important LT performance is to the overall code performance.
- FFT performance is limited by computational intensity, i.e., there isn't enough work to hide the memory latency. Fortran FFT works as well as library FFT for current problem sizes. For larger problems (with a factorization including factors of 8), library routines should see improved performance.

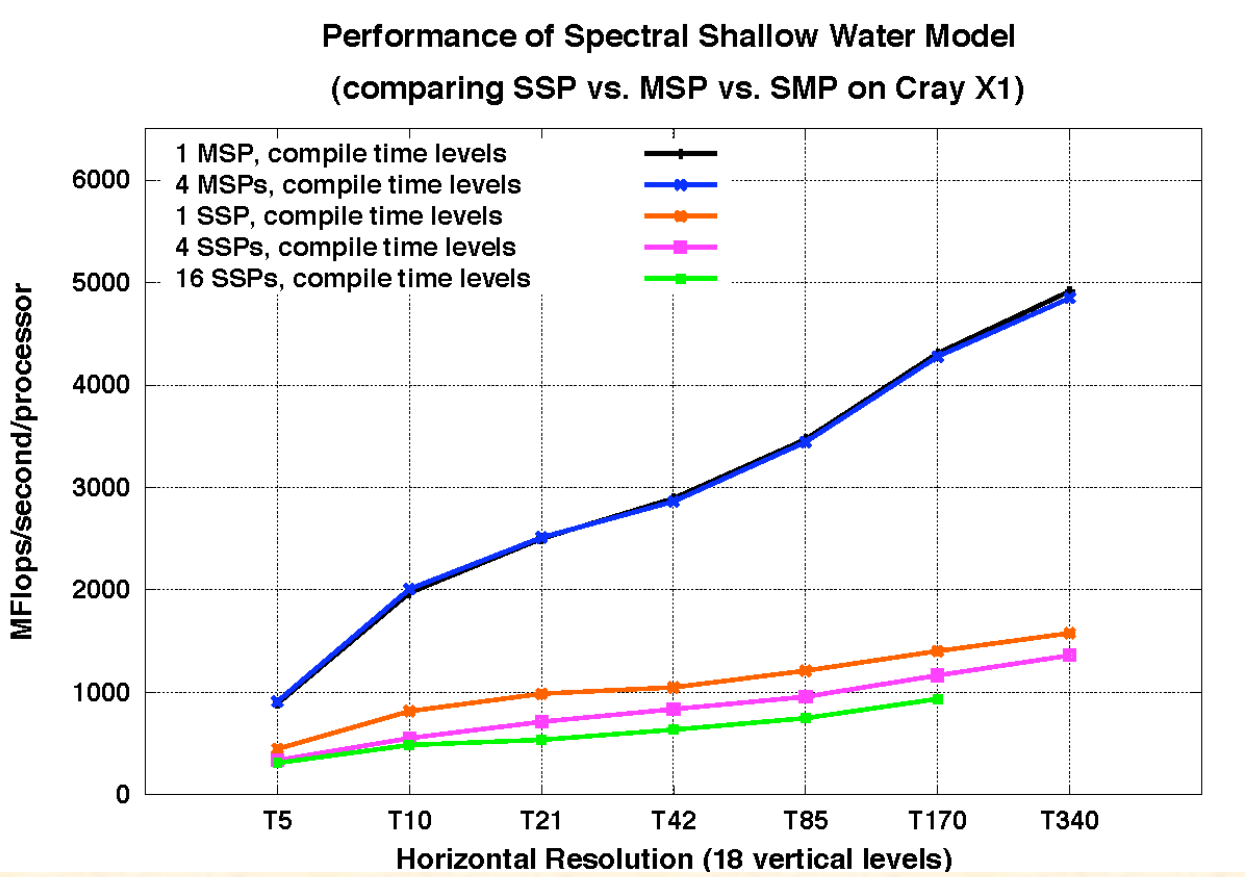
PSTSWM SMP Node Benchmark

Comparing per processor performance when solving same problem simultaneously on all processors in SMP node. X1 MSP data are only data indicating no performance degradation when compared to single processor experiment. Appears to indicate additional advantage to X1 over other systems for scale-up type experiments (for this code).



PSTSWM SMP Node Benchmark

Comparing per processor performance when solving same problem simultaneously on all processors in X1 SMP node. SSP mode sees more contention, and MSP mode achieves better overall throughput for larger problem sizes.

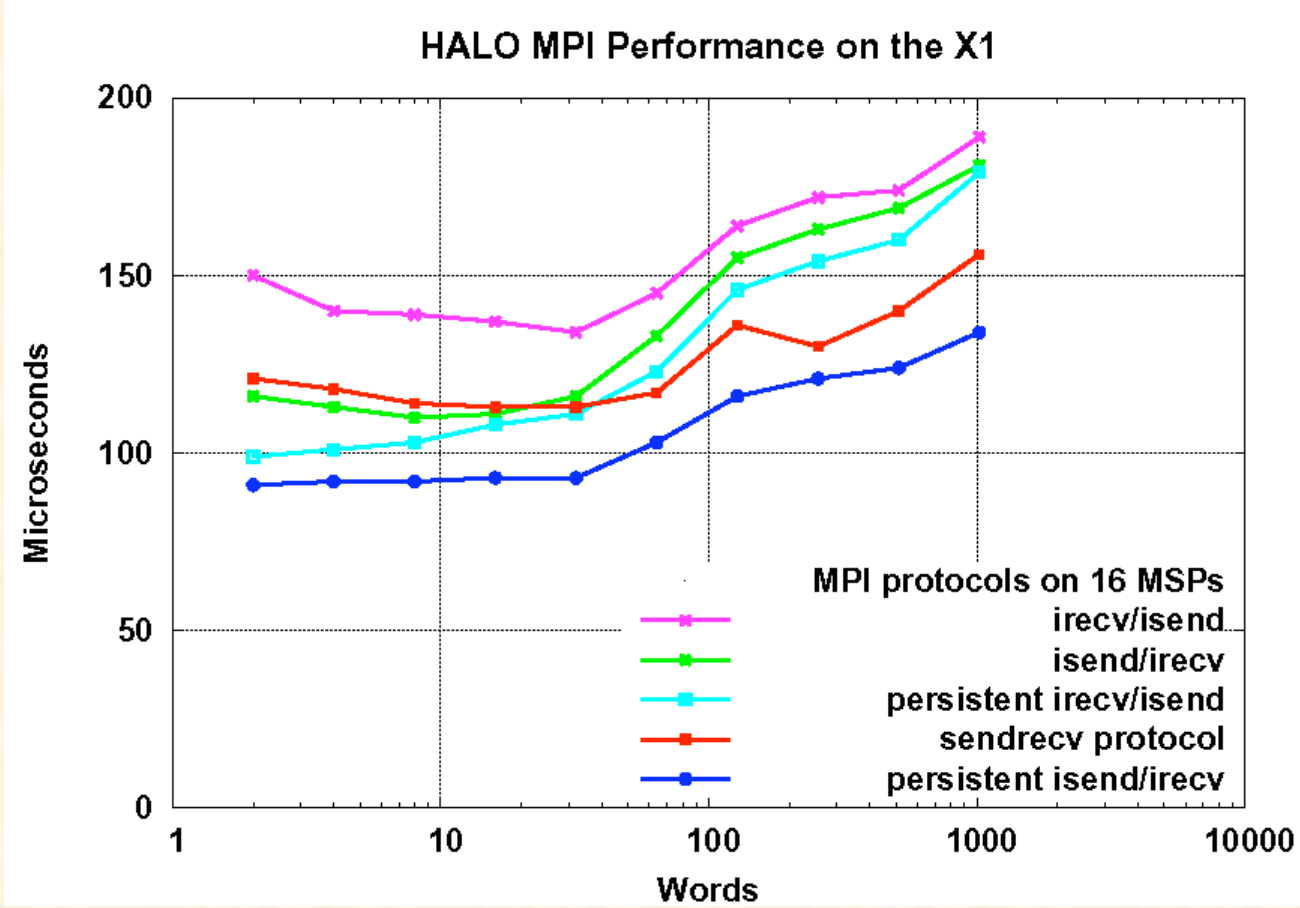


Moral?

- Not all codes are a good match for the Cray X1 architecture. In particular, scalar performance is poor; Code had better vectorize (and/or stream).
- Don't expect good performance without making some attempt at optimization.
- The more information you give the compiler, the better job it can do.
- Large problems typically run better than small problems (relative to nonvector platforms), due to increased vector lengths and good memory subsystem performance.
- Memory access pattern can limit performance on the X1.

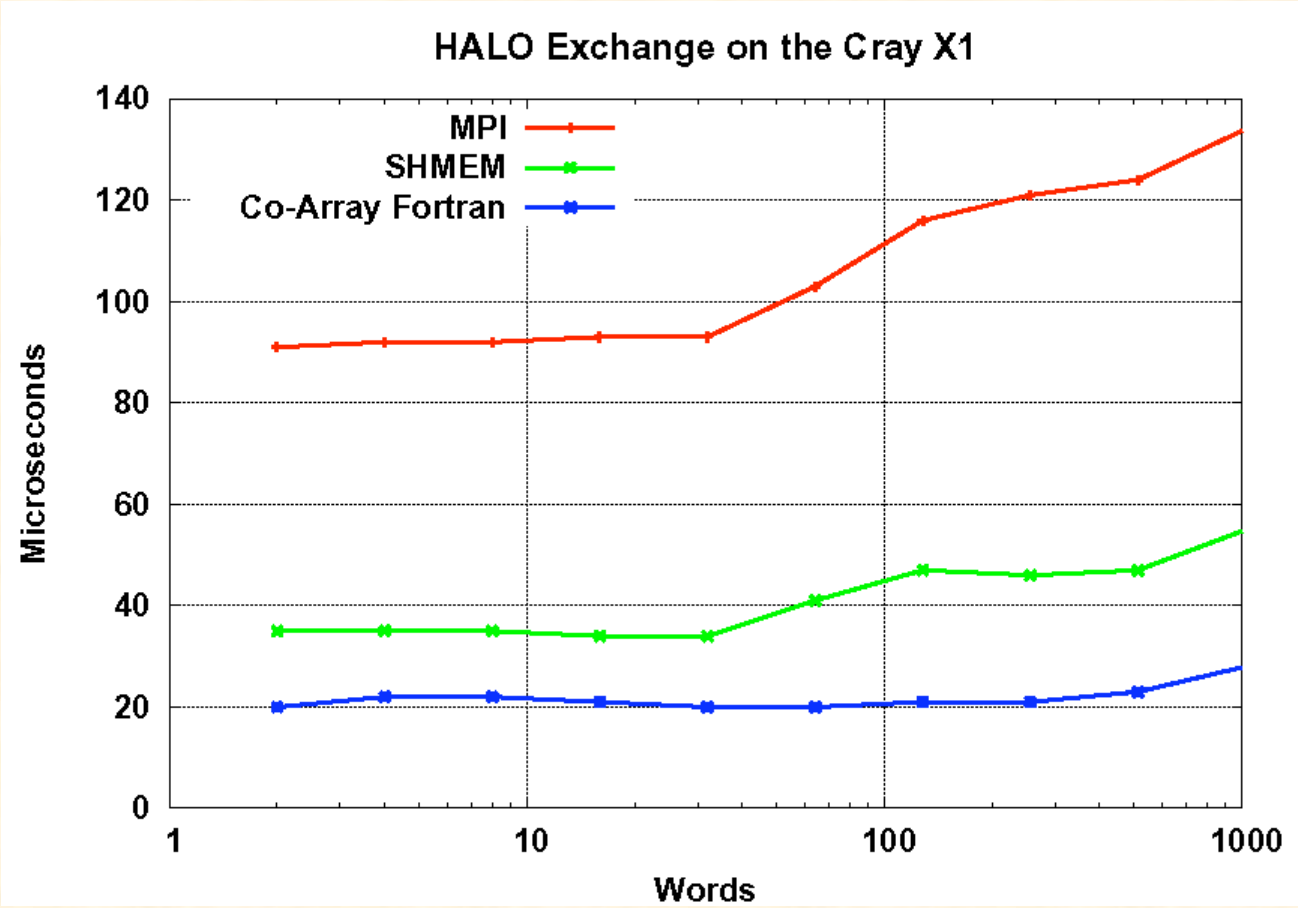
HALO MPI Protocol Comparison

Comparing performance of different MPI implementations of Allan Wallcraft's HALO benchmark on 16 MSPs. Persistent isend/irecv is always best. For codes that can not use persistent commands, MPI_SENDRECV is also a reasonable choice.



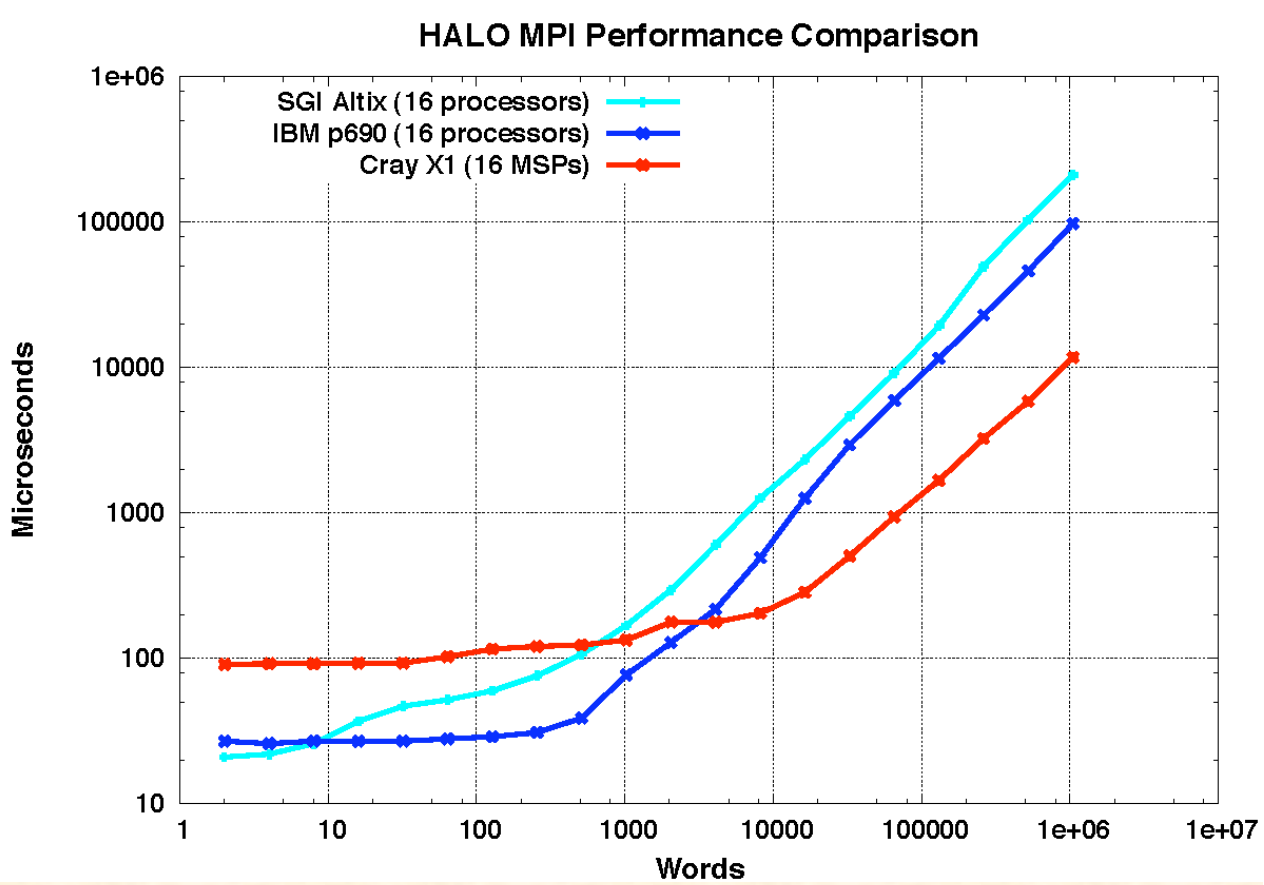
HALO Exchange Paradigm Comparison

Comparing performance of MPI, SHMEM, and Co-Array Fortran implementation HALO benchmark on 16 MSPs. SHMEM and Co-Array Fortran are substantial performance enhancers for this benchmark for small halos.



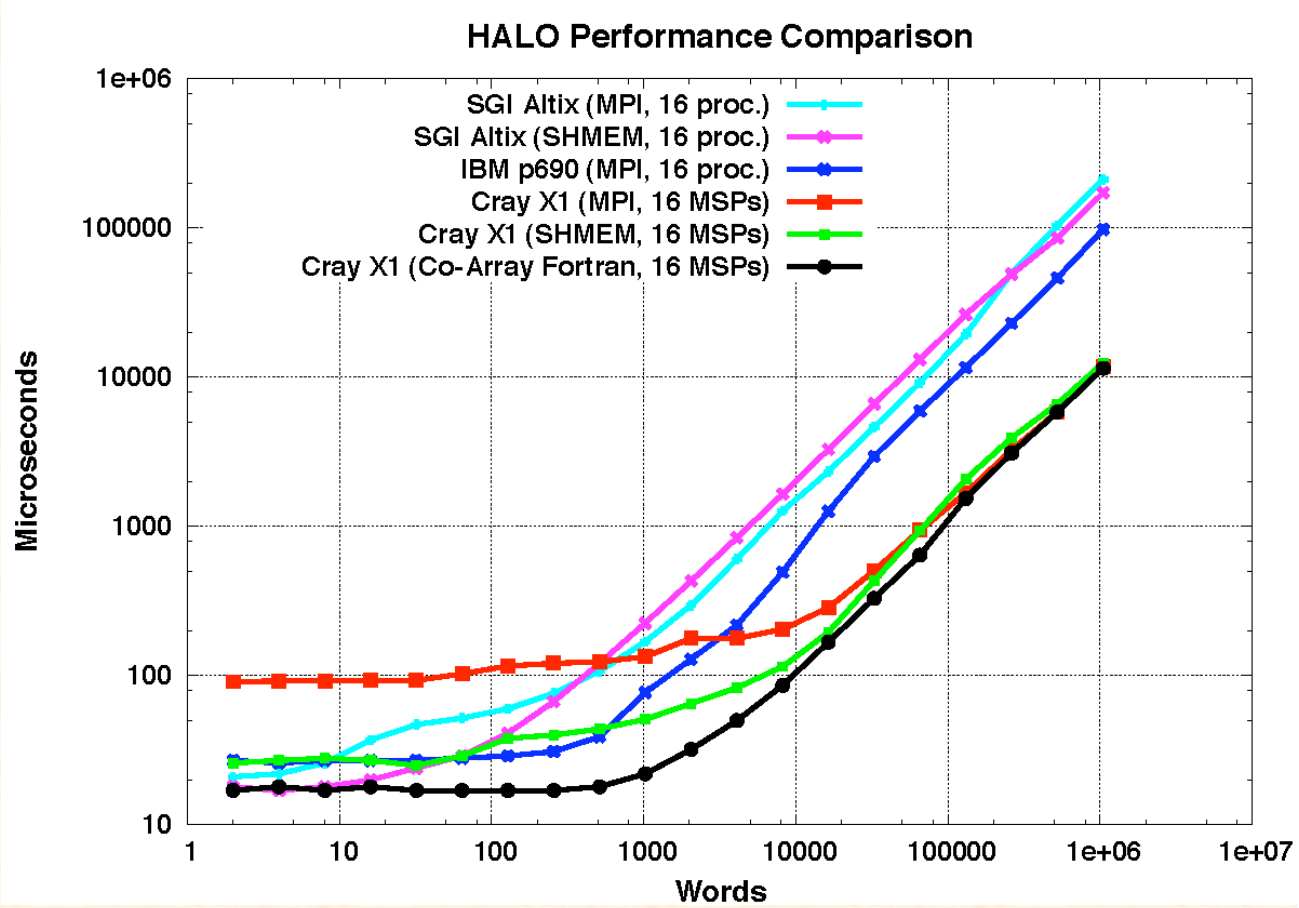
HALO Benchmark

Comparing HALO performance using MPI on 16 MSPs of the Cray X1 and 16 processors of the IBM p690 (within a 32 processor SMP) and the SGI Altix (within a 128 processor SMP). Achievable bandwidth is much higher on the X1. For small halos, the p690 MPI HALO performance is between the X1 SHMEM and Co-Array Fortran HALO performance.



HALO Benchmark

Comparing HALO performance on 16 MSPs of the Cray X1, 16 processors of the IBM p690, and 16 processors of the SGI Altix. Achievable bandwidth is much higher on the X1. For small halos, SHMEM on the X1 ~= MPI on the p690 and Co-Array Fortran on the X1 ~= SHMEM on the Altix.



COMMTEST Benchmark

COMMTEST is a suite of codes that measure the performance of MPI interprocessor communication. In particular, COMMTEST evaluates the impact of communication protocol, packet size, and total message length in a number of “common usage” scenarios. (However, it does not include persistent MPI point-to-point commands among the protocols examined.) It also includes simplified implementations of the SWAP and SENDRECV operators using SHMEM.

COMMTEST Experiments

i-j

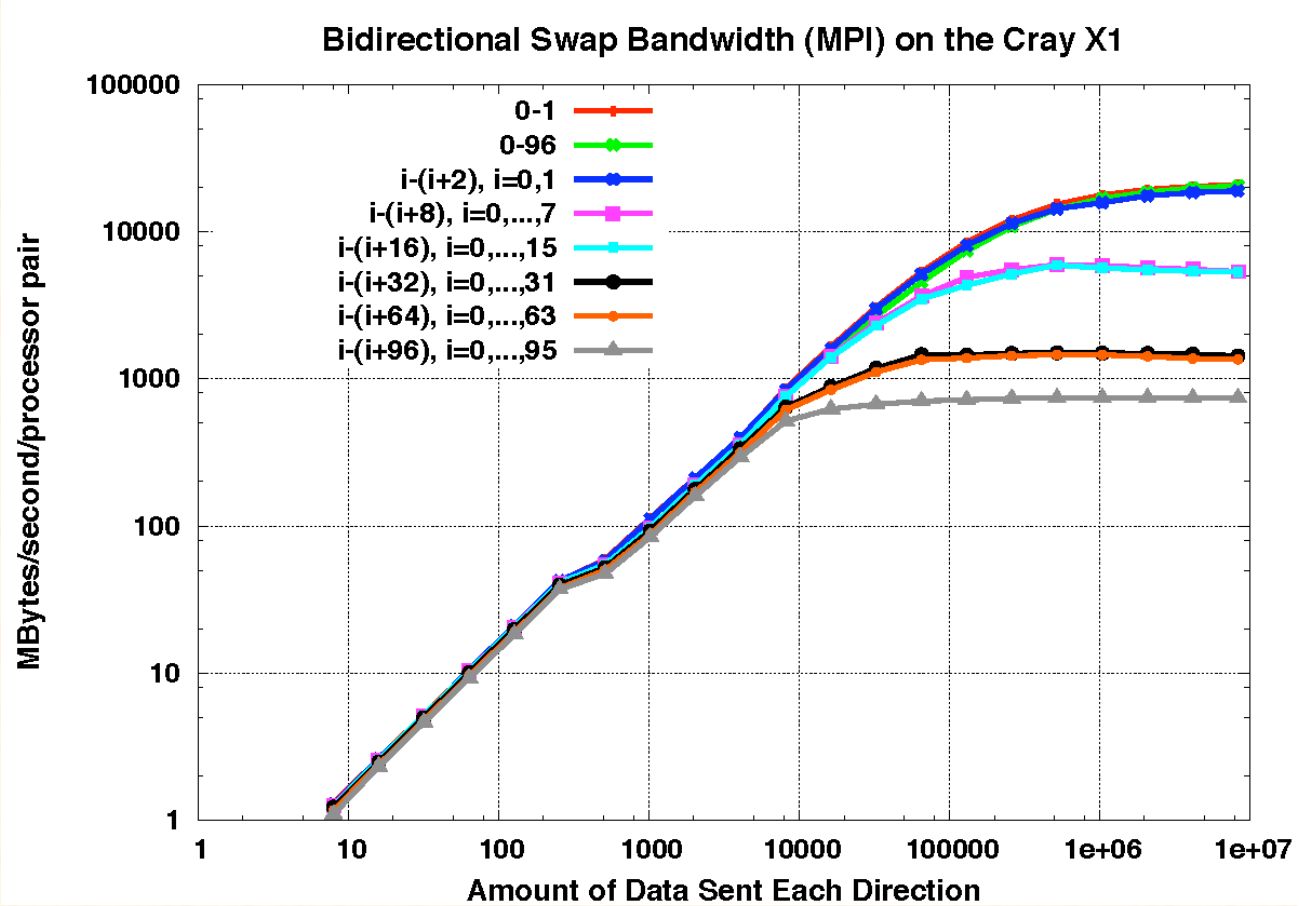
processor i swaps data with processor j. Depending on i and j, this can be within an SMP node or between SMP nodes.

i-(i+j), i=1,n

n processor pairs (i,j) swap data simultaneously. Depending on j, this will be within an SMP node or between SMP nodes (or both).

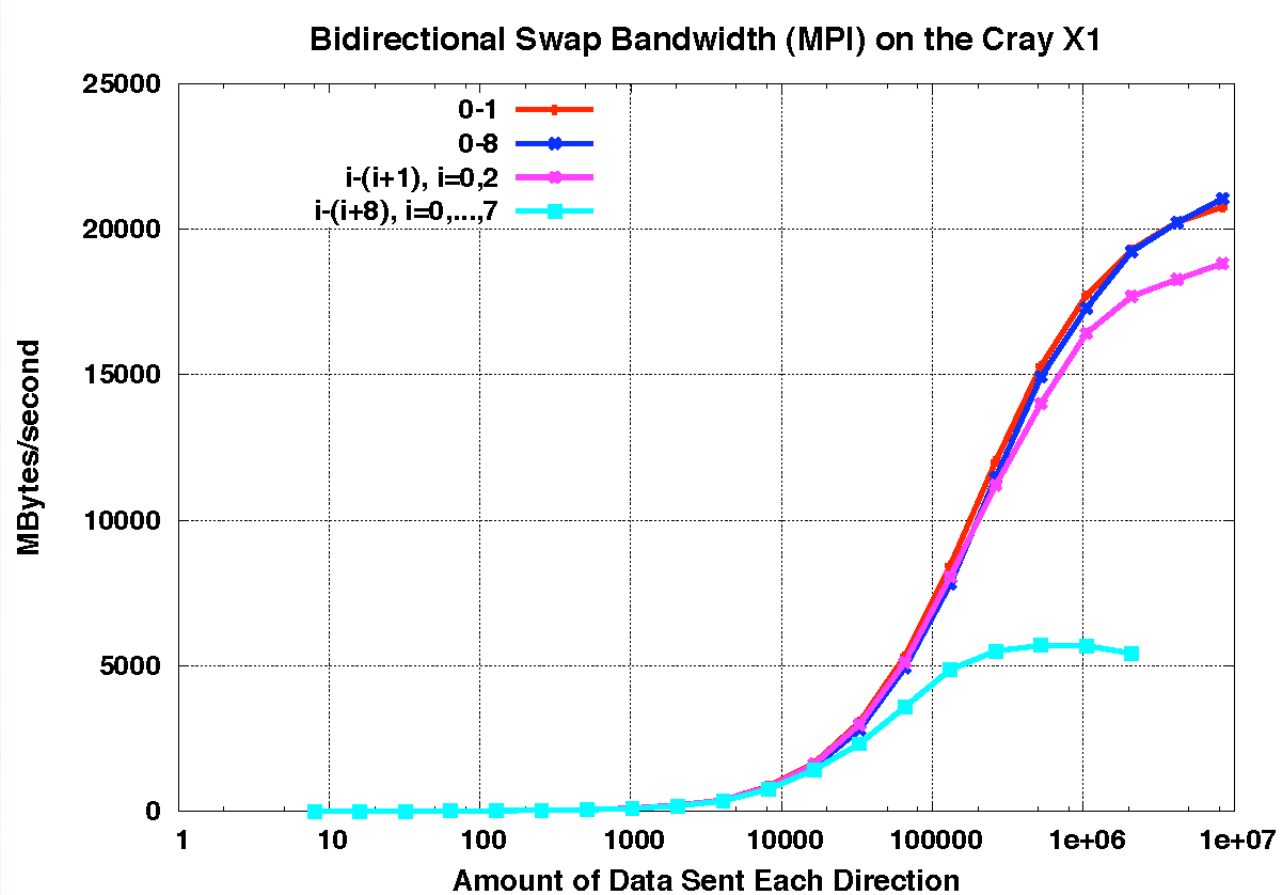
MPI SWAP Evaluation

Comparing performance of SWAP for different communication patterns. Each experiment measures the per processor pair bandwidth when some number of pairs are swapping data simultaneously. For example, i-(i+8),i=0,..,7 means that processor 0 is swapping data with processor 8, processor 1 with processor 9, etc.



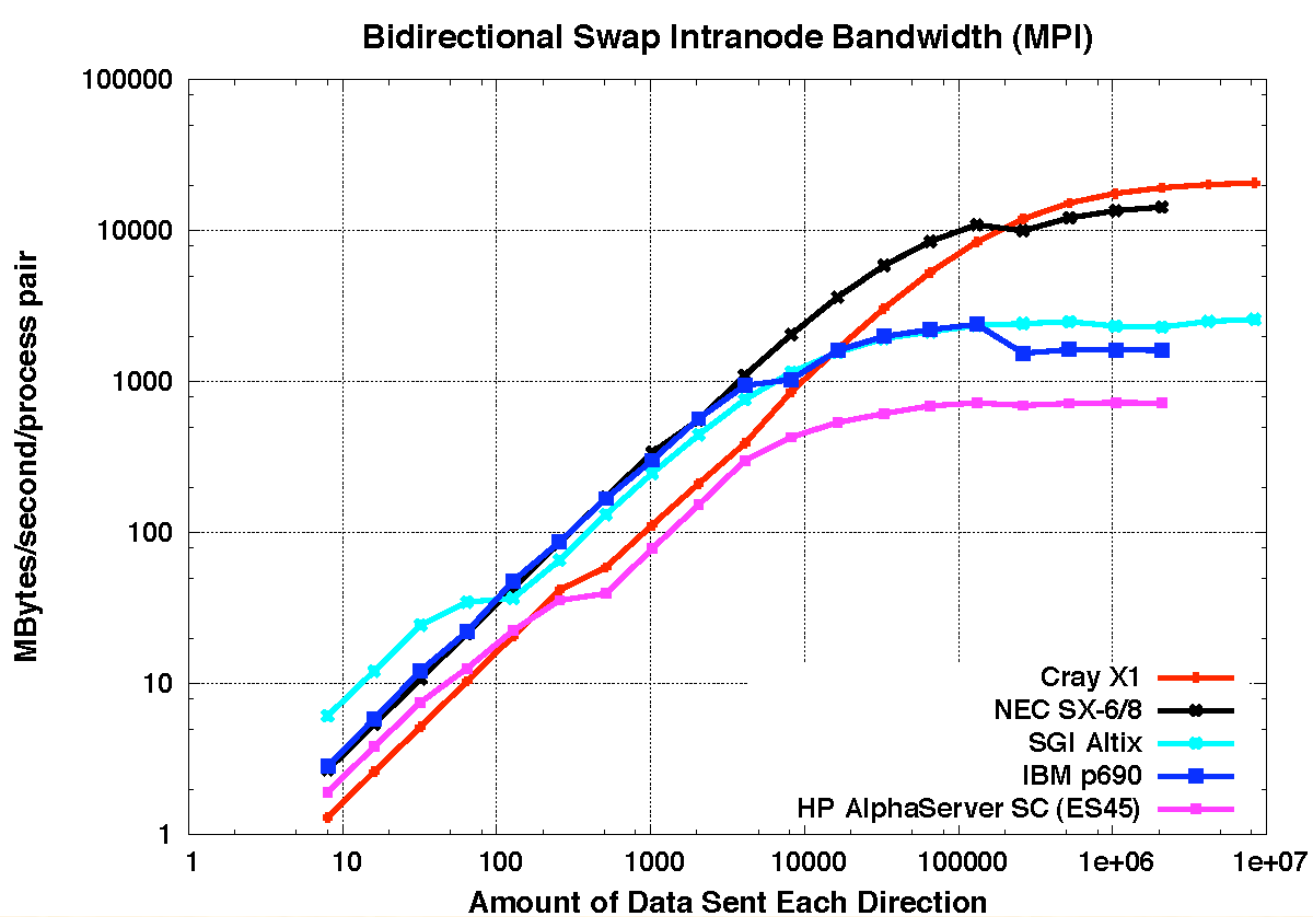
MPI SWAP Evaluation

Comparing performance of SWAP for different communication patterns, plotted on a log-linear scale. The single pair bandwidth has not reached its peak yet, but the two pair experiment bandwidth is beginning to reach its maximum.



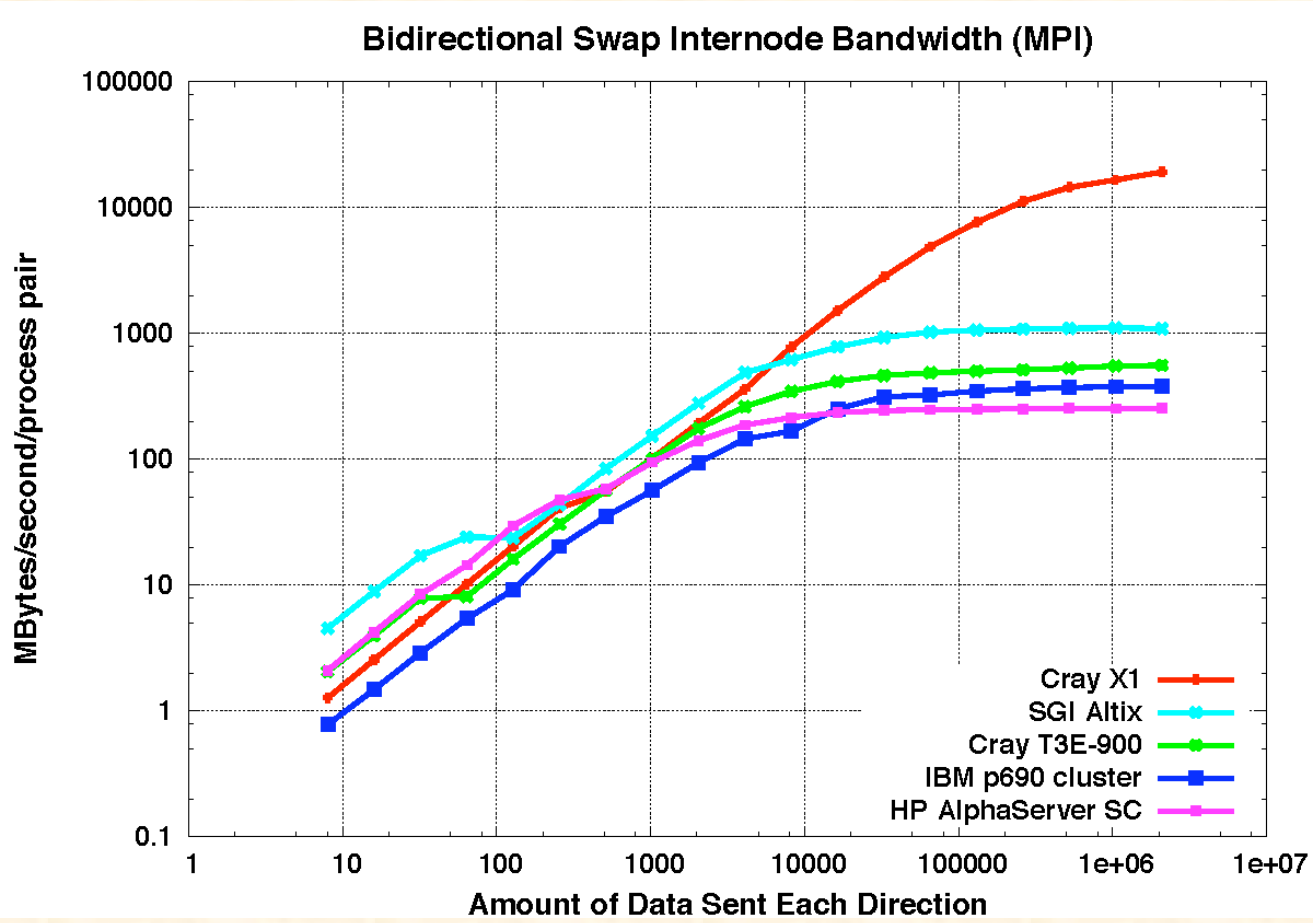
MPI SWAP Platform Comparisons

Comparing performance of SWAP for different platforms. Experiment measures bidirectional bandwidth between two processors in the same SMP node.



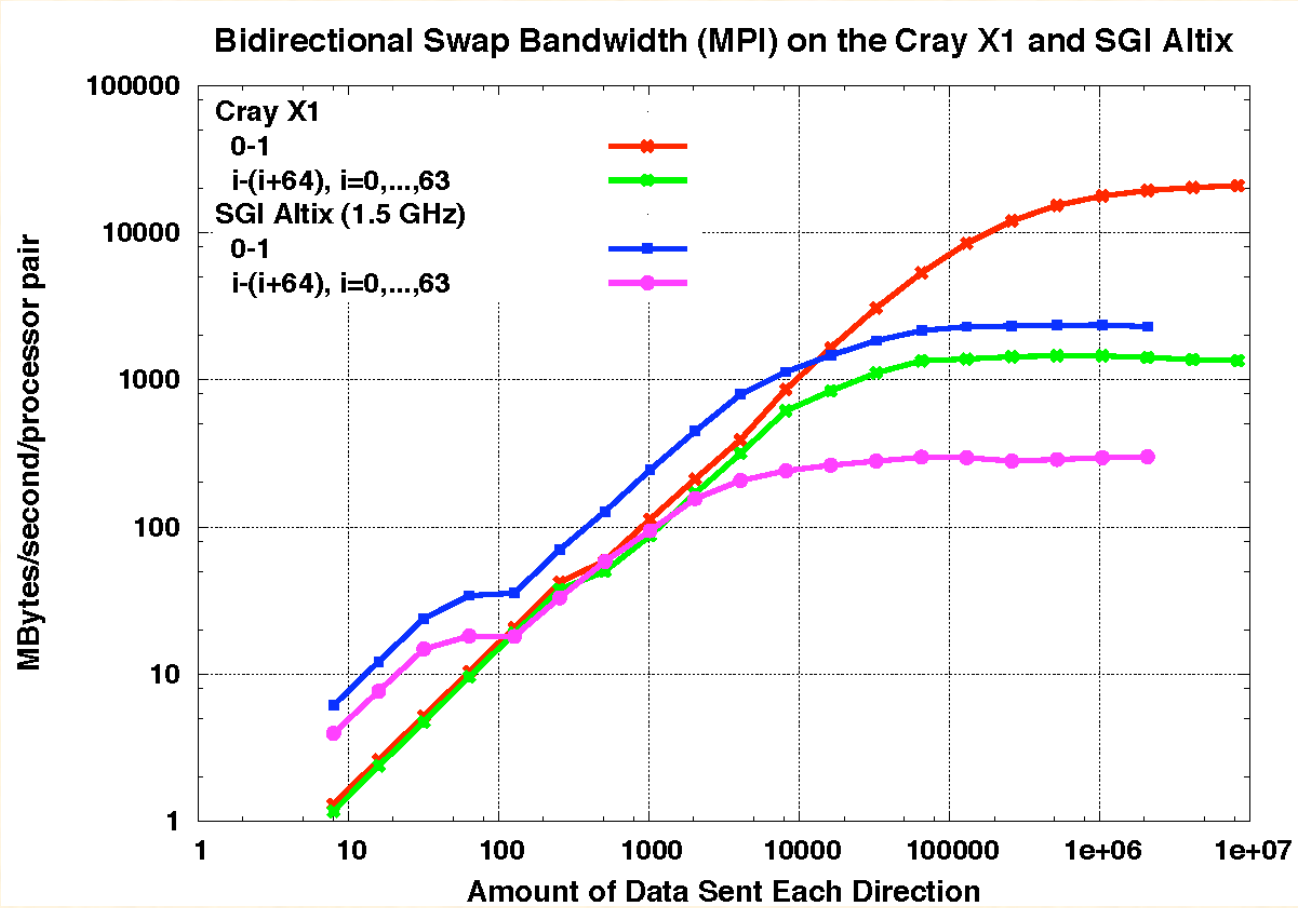
MPI SWAP Platform Comparisons

Comparing performance of SWAP for different platforms. Experiment measures bidirectional bandwidth between two processors in different SMP nodes. For the Altix we used processors 96 and 192. The significant advantage in bandwidth on the X1 is important for a number of important applications.



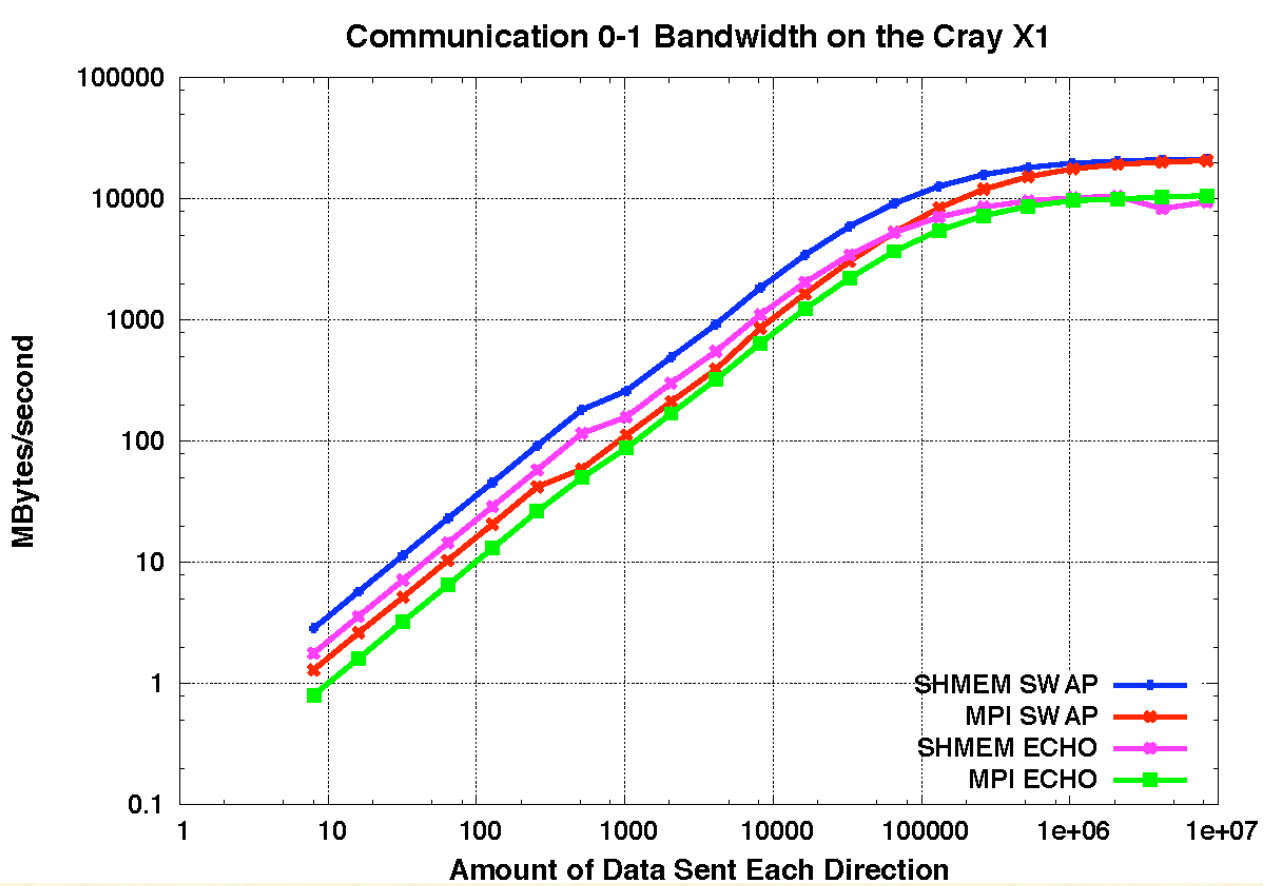
MPI SWAP Contention Evaluation

Comparing performance of SWAP for different communication patterns on the X1 and the Altix. Contention also degrades performance on the Altix, and the X1 retains its advantage.



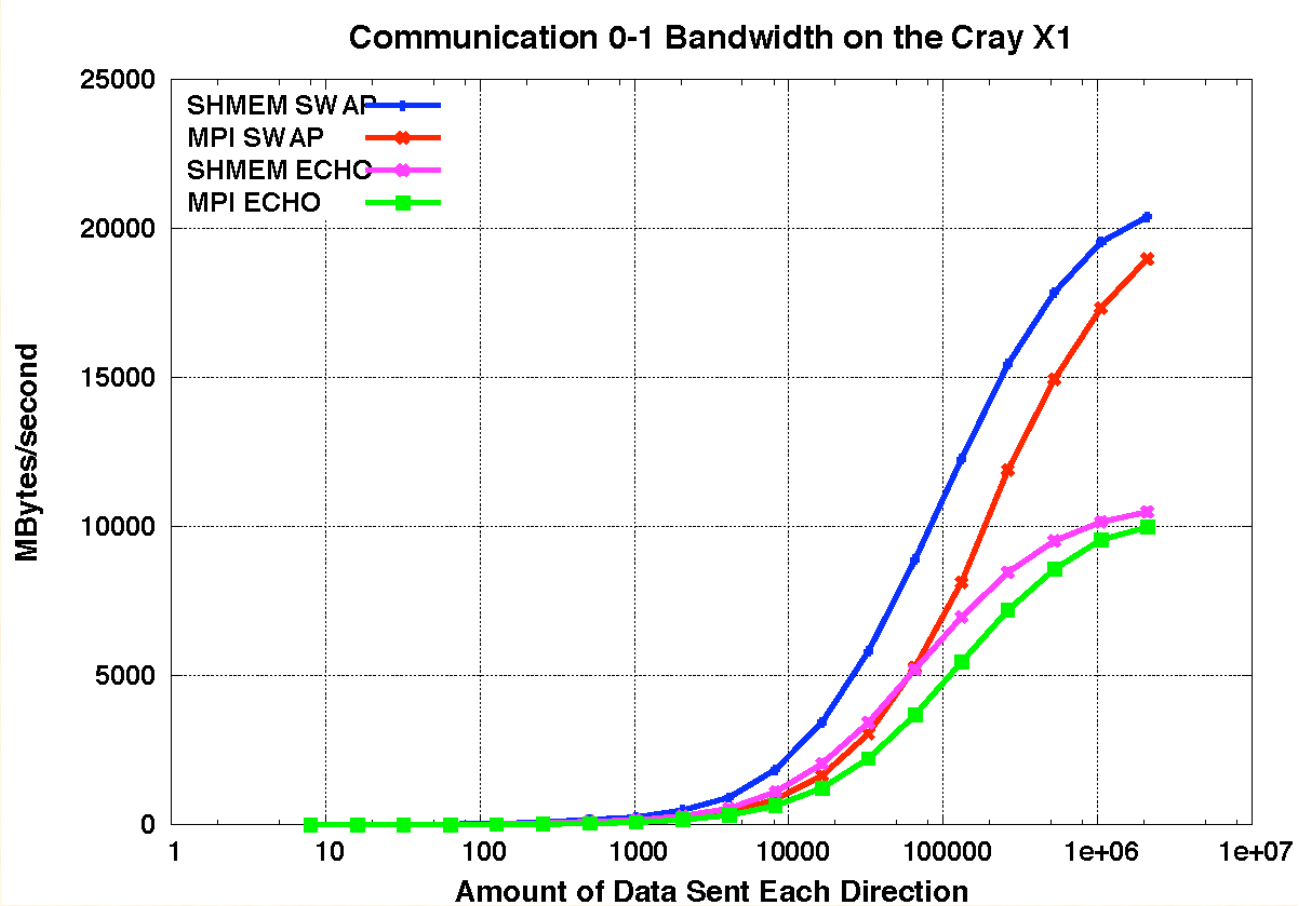
MPI vs. SHMEM 0-1 Comparison on X1

Comparing MPI and SHMEM performance for 0-1 experiment, looking at both SWAP (bidirectional bandwidth) and ECHO (unidirectional bandwidth). SHMEM performance is better for all but the largest messages.



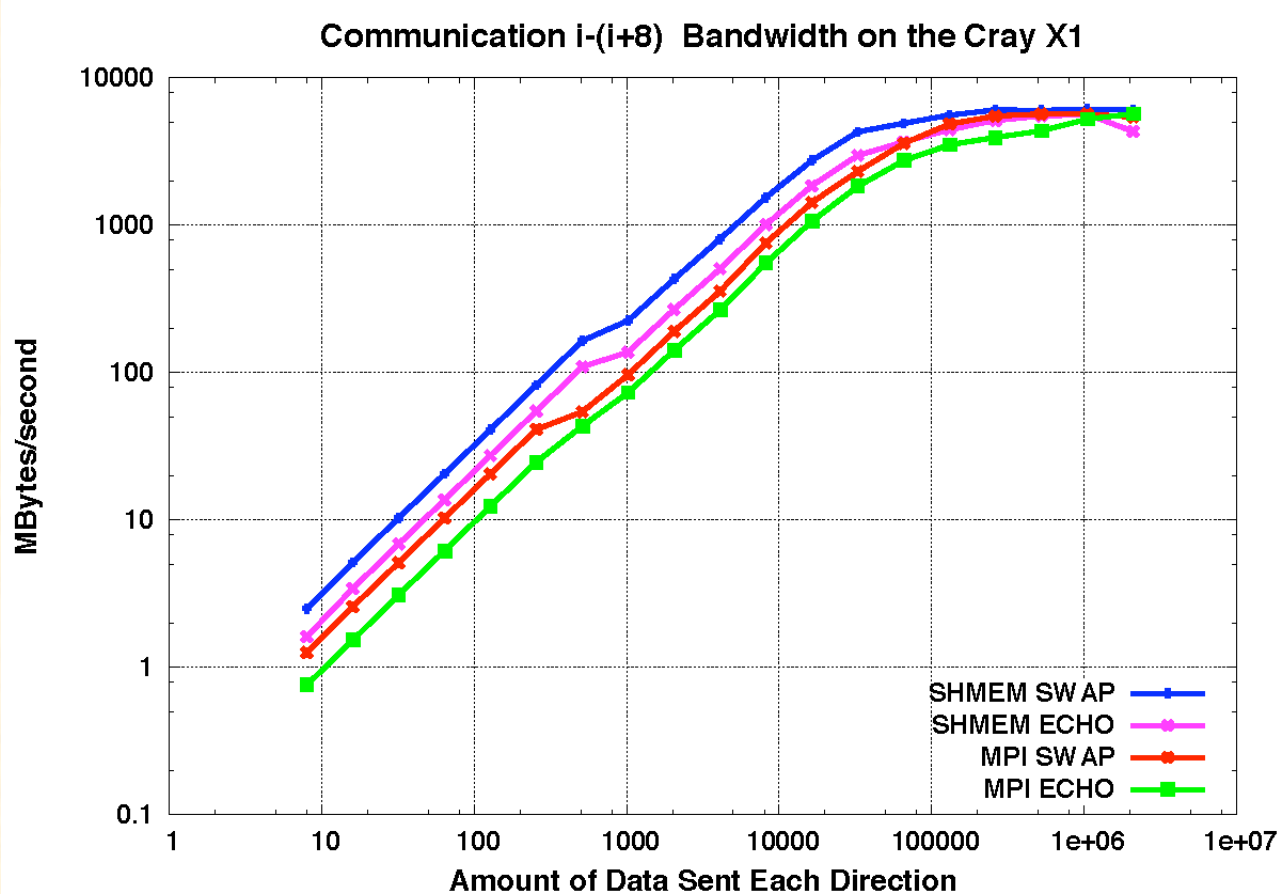
MPI vs. SHMEM 0-1 Comparison on X1

Comparing MPI and SHMEM performance for 0-1 experiment, using a log-linear scale. MPI performance is very near to that of SHMEM for large messages (when using SHMEM to implement two-sided messaging).



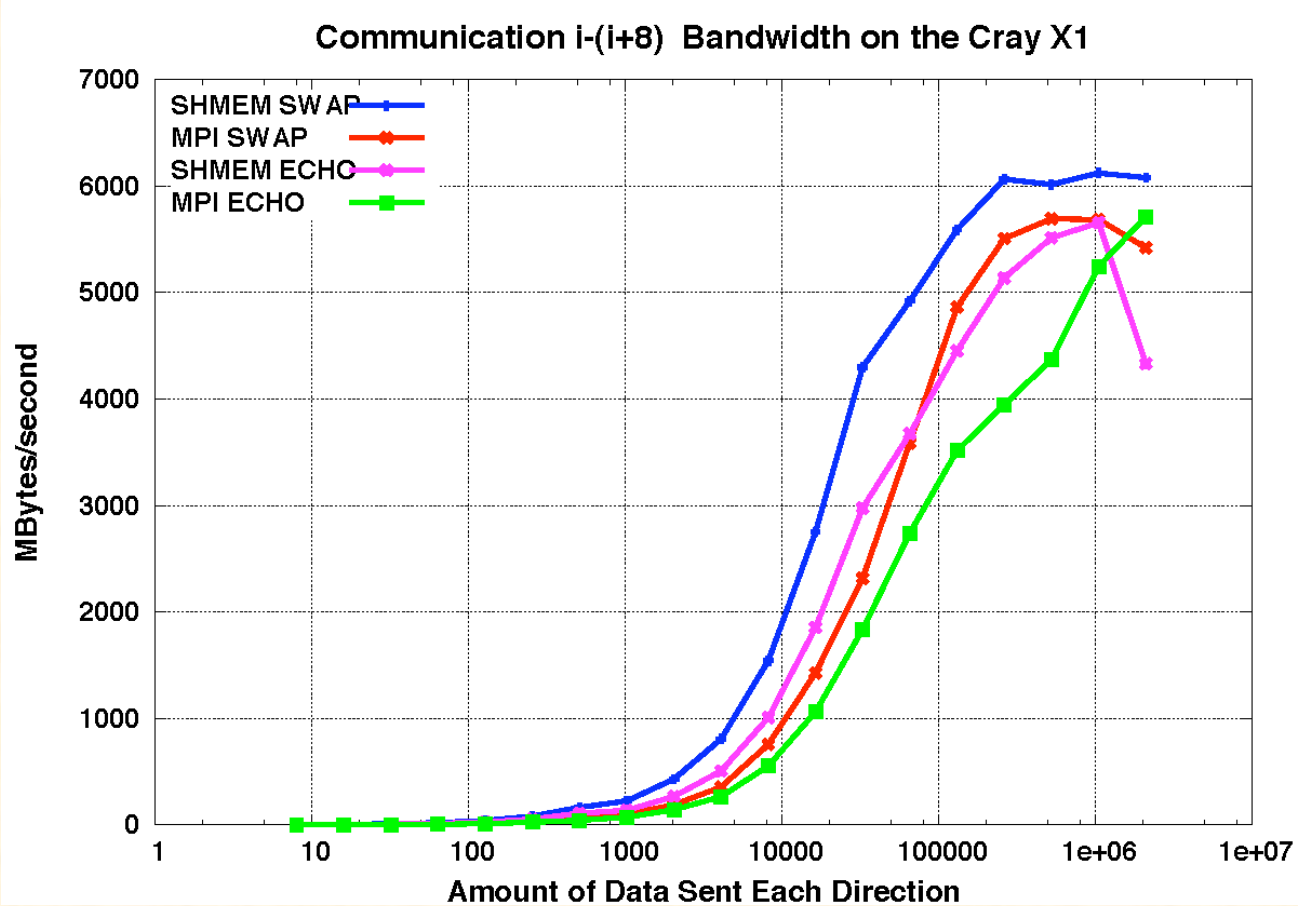
MPI vs. SHMEM i-(i+8) Comparison on X1

Comparing MPI and SHMEM performance for i-(i+8) experiment, looking at both SWAP (bidirectional bandwidth) and ECHO (unidirectional bandwidth). Again, SHMEM performance is better for all but the largest messages.



MPI vs. SHMEM i-(i+8) Comparison on X1

Comparing MPI and SHMEM performance for i-(i+8) experiment, using a log-linear scale. MPI performance is very near to that of SHMEM for large messages (when using SHMEM to implement two-sided messaging). For the largest message sizes, MPI ECHO bandwidth exceeds MPI SWAP bandwidth.



Moral?

- The MPI protocol used can impact performance.
- MPI (and SHMEM) support bidirectional communication.
- MPI latency is mediocre at the current time. However, latency for Co-Array Fortran or SHMEM is excellent. (There is reason to believe that that MPI small message performance will improve.)
- MPI bandwidth is excellent for individual processor pairs. Performance degradation due to contention is well-behaved and is significant for the examined communication pattern only for messages > 8KBytes on the current system.
- MPI latency and bandwidth are relatively insensitive to distance between communicating processors on the current system.

Application Performance Diagnosis

- POP Ocean code
  - Performance impact of vectorization
  - Performance impact of MPI tuning
  - Performance impact of Co-array Fortran
- PSTSWM spectral transform benchmark
  - Algorithm comparison
  - Performance impact of load imbalance
  - Performance impact of memory (mis)alignment
- CAM Atmospheric code
  - Performance impact of load balancing
- EVH1 hydronamics code
  - Performance impact of vector length
  - Performance tradeoffs between SSP and MSP mode

Parallel Ocean Program (POP)

- Developed at Los Alamos National Laboratory. Used for high resolution studies and as the ocean component in the Community Climate System Model (CCSM)
- Ported to the Earth Simulator by Dr. Yoshikatsu Yoshida of the Central Research Institute of Electric Power Industry (CRIEPI).
- Initial port to the Cray X1 by John Levesque of Cray, using Co-Array Fortran for conjugate gradient solver.
- X1 and Earth Simulator ports merged and modified by Pat Worley and Trey White of Oak Ridge National Laboratory.
- “Optimization on the X1 ongoing.”

POP Experiment Particulars

- Two primary computational phases
  - Baroclinic: 3D with limited nearest-neighbor communication; scales well.
  - Barotropic: dominated by solution of 2D implicit system using conjugate gradient solves; scales poorly.
- One fixed size benchmark problem
  - One degree horizontal grid (“by one” or “x1”) of size 320x384x40.
- Domain decomposition determined by grid size and 2D virtual processor grid. Results for a given processor count are the best observed over all applicable processor grids.

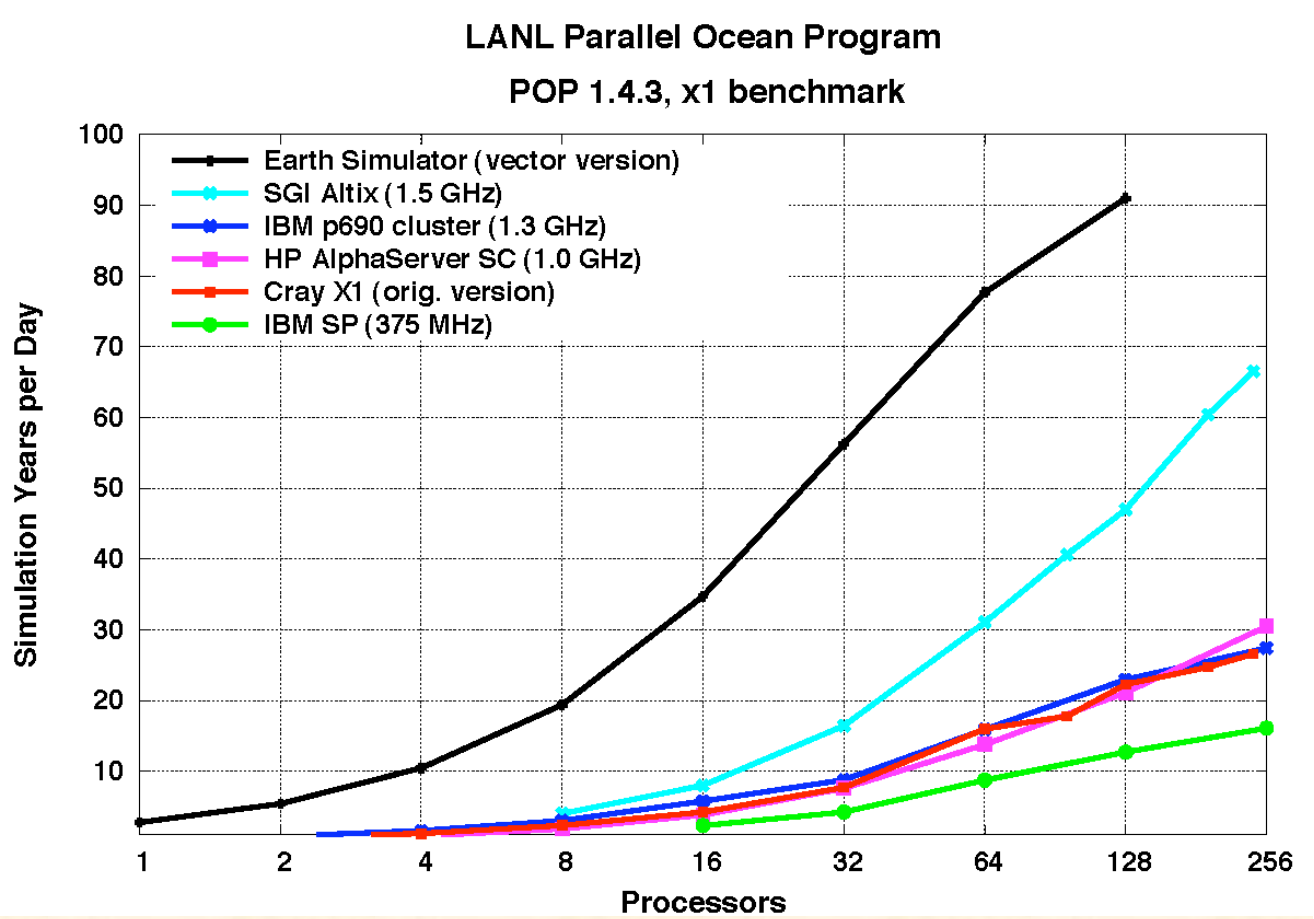
POP Performance Tools and Techniques

- Benchmarking
  - Platform comparisons
  - Scaling studies
- User instrumentation (timers)
- Loopmarks
- Instrumented Communication Library (MPICL)
- Performance Visualization Tool (ParaGraph)
- Performance Analysis Tool (PAT)
- (OS instrumentation)

POP Platform Comparison: Initial

Comparing performance and scaling across platforms.

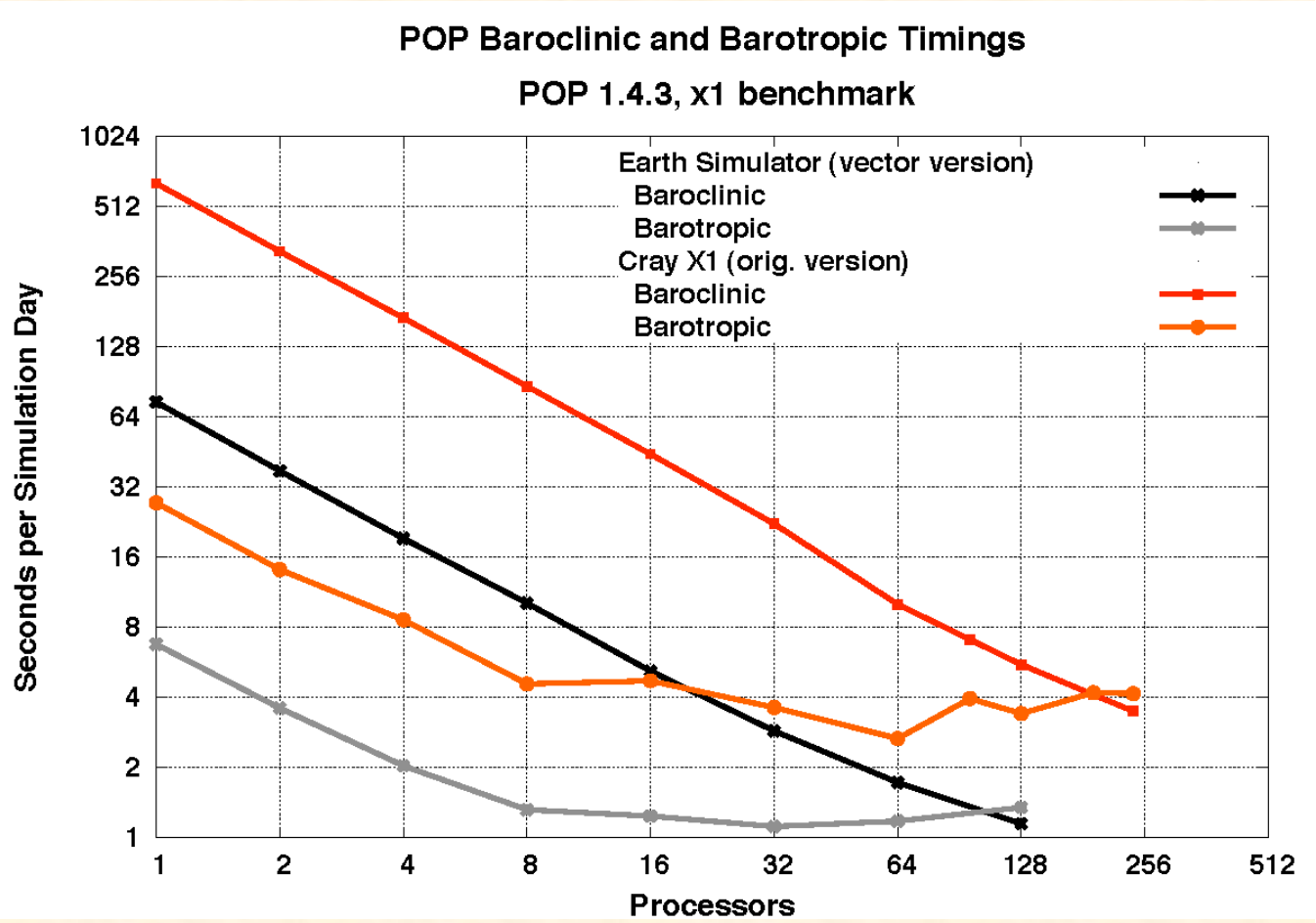
- Earth Simulator results courtesy of Dr. Y. Yoshida of the Central Research Institute of Electric Power Industry
- IBM SP results courtesy of Dr. T. Mohan of Lawrence Berkeley National Laboratory
- X1 results (using standard distribution) are indistinguishable from those collected on IBM and HP systems.



Initial Performance Diagnosis vs. ES40

Baroclinic phase on the ES40 (running the ES40 port of POP) is 8 times faster than on the X1 (running the standard distribution of POP)

Barotropic phase on the ES40 (running the ES40 port of POP) is 4 times faster than on the X1 (running the standard distribution of POP)



POP Vectorization

- Began with port to Earth Simulator
  - 701 lines replaced (by 1168 lines), out of 45000 lines
  - Over half of the ES modifications (~400 lines) do not change performance on the X1 significantly: e.g., replacement of F90 where, merge, eoshift, … by F77 equivalents.
- Modified two (previously modified) routines to improve performance of ES port on the X1
  - Number of lines replaced in original version approximately the same for the ES and X1 versions currently

POP Vectorization

Loopmarks from original version

```
750.  1------------<       do j=jphys_b,jphys_e
751.  1 2----------<       do i=iphys_b,iphys_e
752.  1 2
753.  1 2                  !*** solve tridiagonal system at each grid point
754.  1 2
755.  1 2                  a = afac_u(1)*VVC(i,j,1)
756.  1 2                  d = hfac_u(1) + a
757.  1 2                  e(1) = a/d
758.  1 2                  b = hfac_u(1)*e(1)
759.  1 2                  f1(1) = hfac_u(1)*UVEL(i,j,1,newtime)/d
760.  1 2                  f2(1) = hfac_u(1)*VVEL(i,j,1,newtime)/d
```

POP Vectorization

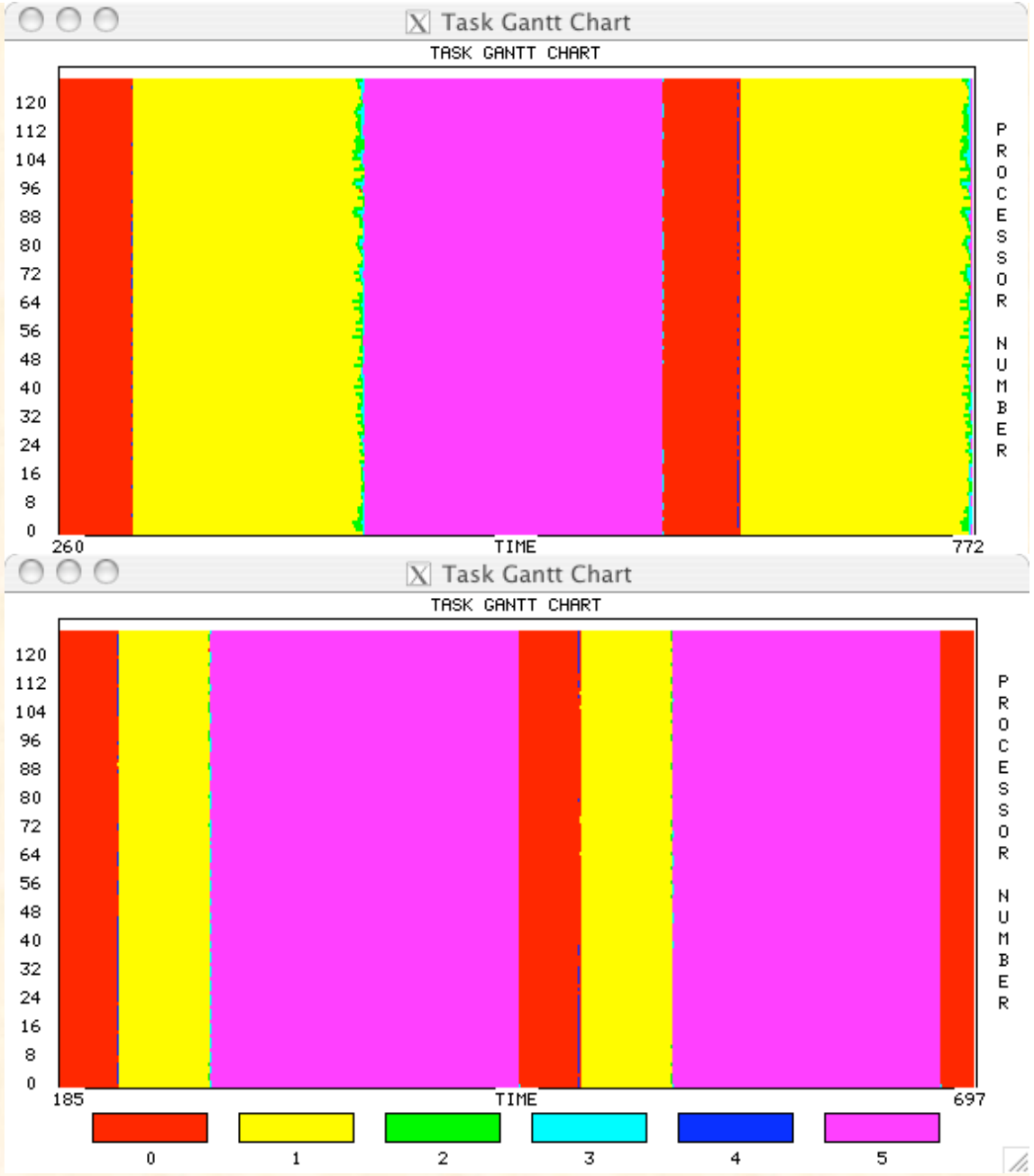
Loopmarks from vector version

```
792.  M------------<       do j=1,jmt
793.  M Vs---------<          do i=1,imt
794.  M Vs
795.  M Vs                    A(i,j)    = afac_u(1)*VVC(i,j,1)
796.  M Vs                    D(i,j)    = hfac_u(1) + A(i,j)
797.  M Vs                    E(i,j,1)  = A(i,j)/D(i,j)
798.  M Vs                    B(i,j)    = hfac_u(1)*E(i,j,1)
799.  M Vs                    F1(i,j,1) = hfac_u(1)*UVEL(i,j,1,newtime)/D(i,j)
800.  M Vs                    F2(i,j,1) = hfac_u(1)*VVEL(i,j,1,newtime)/D(i,j)
801.  M Vs
802.  M Vs--------->          end do
803.  M------------>       end do
```

Performance Impact of Vectorization

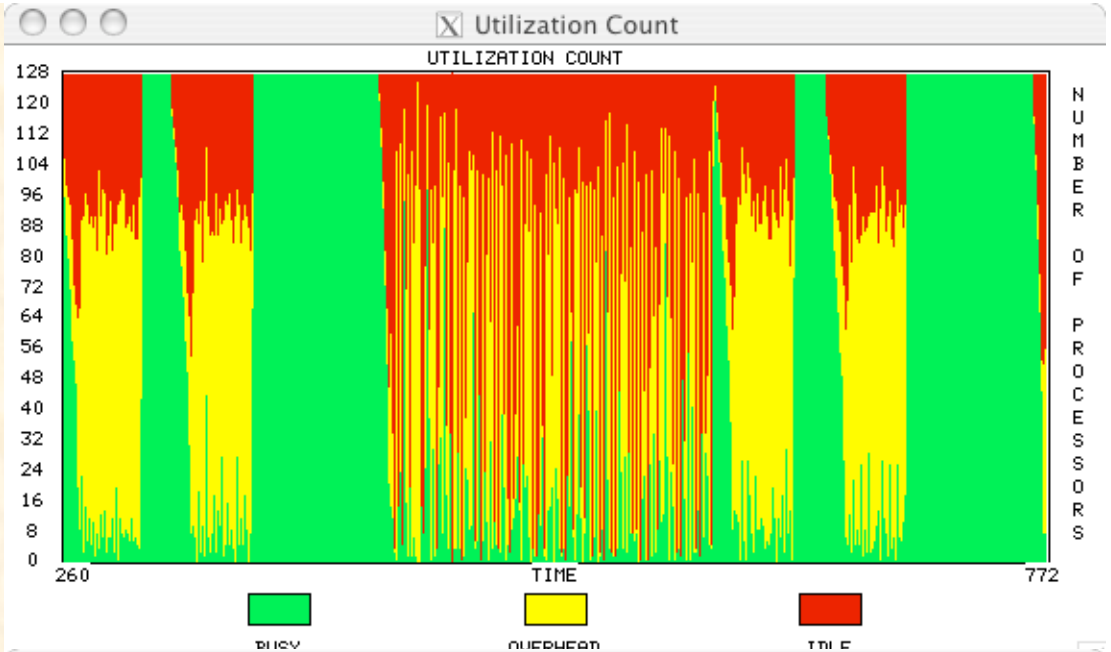
Task Gannt chart before and after code vectorization for 128 MSP run:

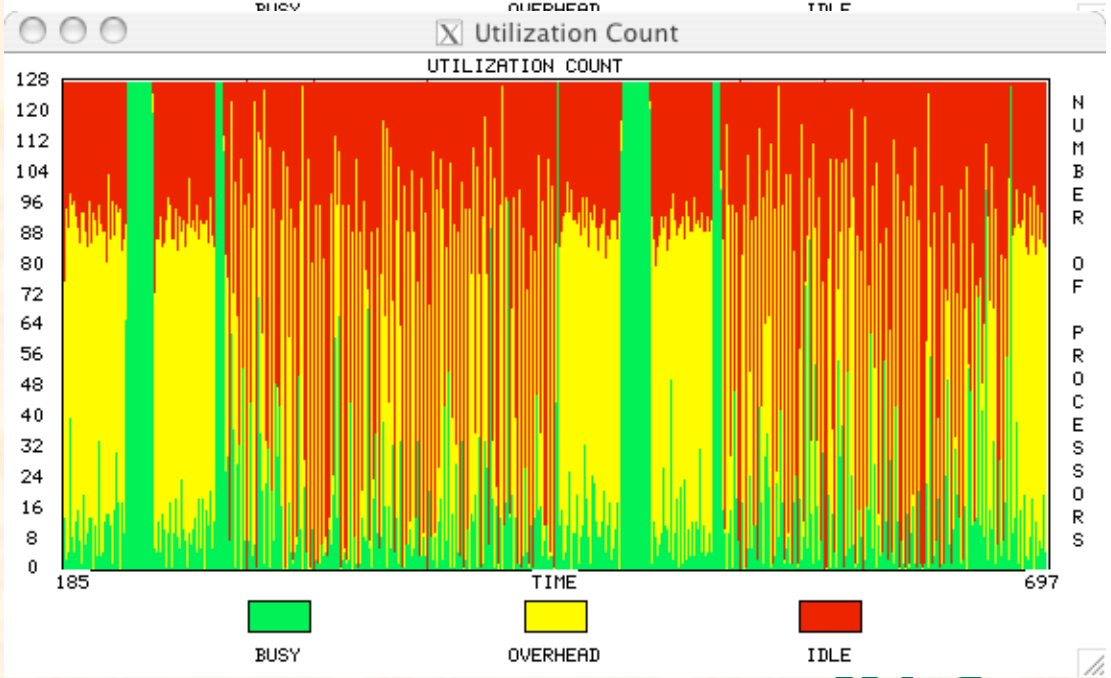
0: (primarily tracer updates)
1: baroclinic
2: baroclinic boundary update
3: barotropic (excl. solver)
4: barotropic boundary update
5: barotropic solver



Performance Impact of Vectorization

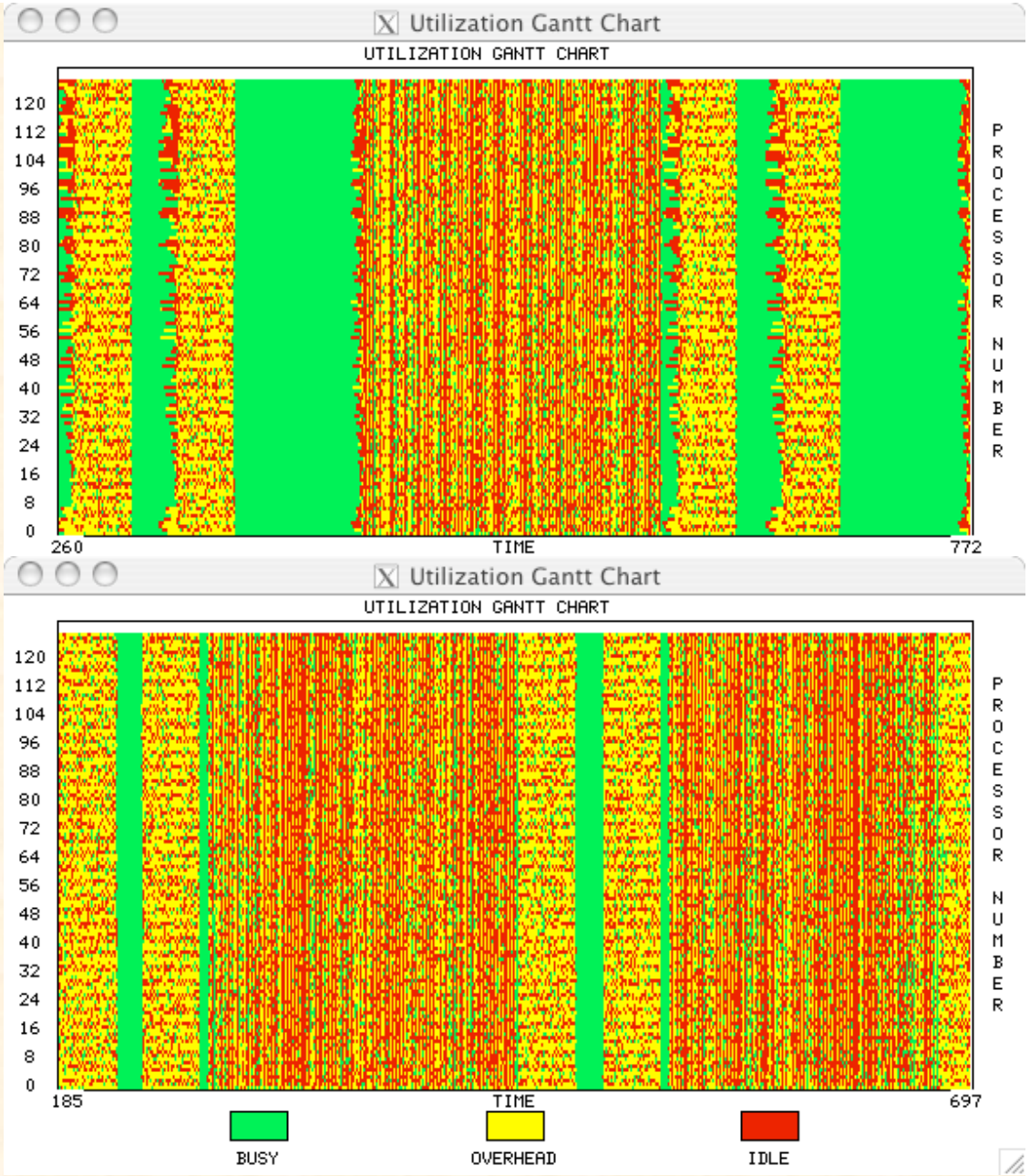
Compute (Busy) / Communicate (Overhead and Idle) utilization graph before and after code vectorization for 128 MSP run.





Performance Impact of Vectorization

Compute (Busy) / Communicate (Overhead and Idle) Gannt chart before and after code vectorization for 128 MSP run.



POP MPI Optimization

- Used Earth Simulator port unchanged
  - 125 lines replaced (by 233 lines); one new (140 line) routine added.
  - Five routines modified to replace irecv/isend logic with isend/recv logic. (The performance of POP on the X1 is not sensitive to this. Neither approach degrades overall performance.)
  - Three routines modified to replace communication of halo regions using derived datatypes with packing/unpacking message buffers and using standard datatypes. Routines called in barotropic phase.
  - New routine added to block communication, replacing communication of many small messages by a few large messages. Routine called in baroclinic phase and in baroclinic_correct_adjust (updating tracers).

POP MPI Optimization

MPI from original version (halo update)

```
call MPI_IRECV(XOUT(1,1), 1, mpi_ew_type, nbr_west,…)
call MPI_IRECV(XOUT(iphys_e+1,1), 1, mpi_ew_type, nbr_east, …)
call MPI_ISEND(XOUT(iphys_e+1-num_ghost_cells,1), 1, mpi_ew_type, …)
call MPI_ISEND(XOUT(iphys_b,1), 1, mpi_ew_type, nbr_west,…)
call MPI_WAITALL(4, request, status, ierr)

call MPI_IRECV(XOUT(1,jphys_e+1), 1, mpi_ns_type, nbr_north, …)
call MPI_IRECV(XOUT(1,1), 1, mpi_ns_type, nbr_south,…)
call MPI_ISEND(XOUT(1,jphys_b), 1, mpi_ns_type, nbr_south,…)
call MPI_ISEND(XOUT(1,jphys_e+1-num_ghost_cells), 1,mpi_ns_type, …)
call MPI_WAITALL(4, request, status, ierr)
```

using derived types.

POP MPI Optimization

MPI from MPI optimized version (halo update)

(... copy XOUT into send buffers ...)

```
call MPI_ISEND(buffer_east_snd, buf_len_ew, MPI_DOUBLE_PRECISION,...)
call MPI_ISEND(buffer_west_snd, buf_len_ew, MPI_DOUBLE_PRECISION,...)
call MPI_RECV(buffer_west_rcv,  buf_len_ew, MPI_DOUBLE_PRECISION,...)
call MPI_RECV(buffer_east_rcv,  buf_len_ew, MPI_DOUBLE_PRECISION,...)
call MPI_WAITALL(2, request, status_wait, ierr)
```

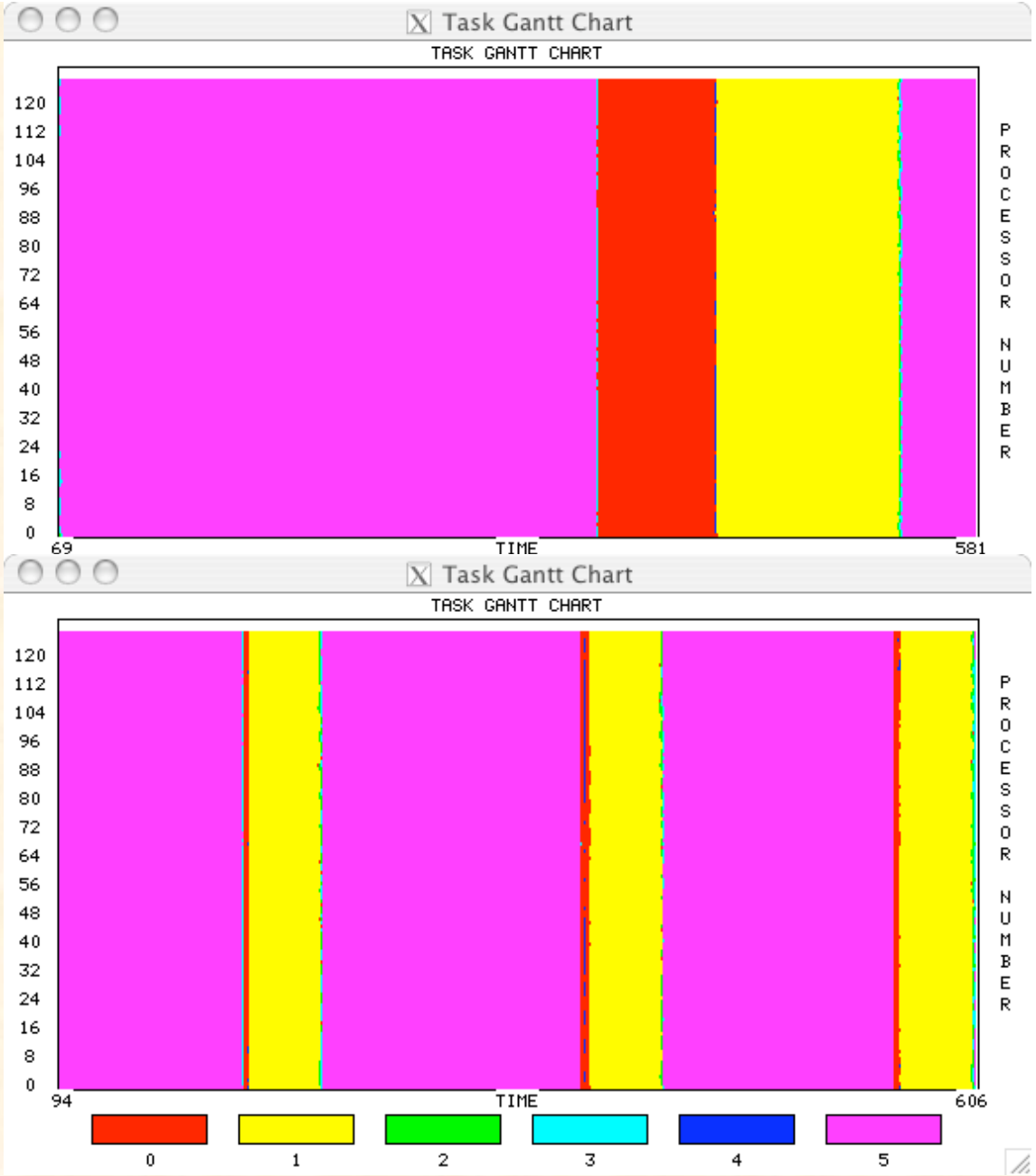
(... copy receive buffers into XOUT ...)

```
call MPI_ISEND(XOUT(1,jphys_e+1-num_ghost_cells), buf_len_ns, MPI_DOUBLE_PRECISION,...)
call MPI_ISEND(XOUT(1,jphys_b), buf_len_ns, MPI_DOUBLE_PRECISION, ...)
call MPI_RECV(XOUT(1,jphys_e+1), buf_len_ns, MPI_DOUBLE_PRECISION, ...)
call MPI_RECV(XOUT(1,1), buf_len_ns, MPI_DOUBLE_PRECISION,...)
call MPI_WAITALL(2, request, status_wait, ierr)
```

Performance Impact of MPI Tuning

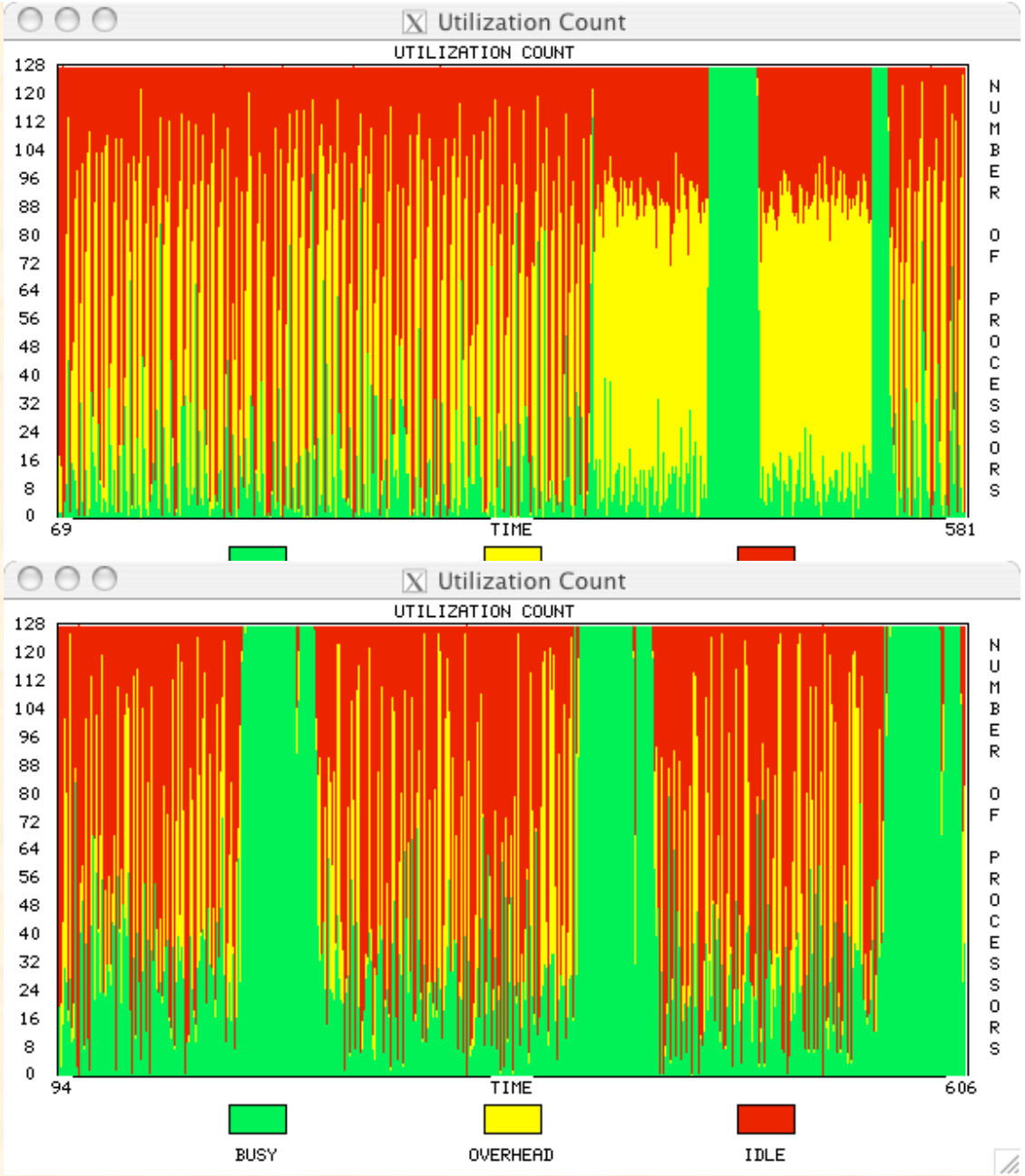
Task Gannt chart before and after MPI optimization for 128 MSP run:

0: (primarily tracer updates)
1: baroclinic
2: baroclinic boundary update
3: barotropic (excl. solver)
4: barotropic boundary update
5: barotropic solver



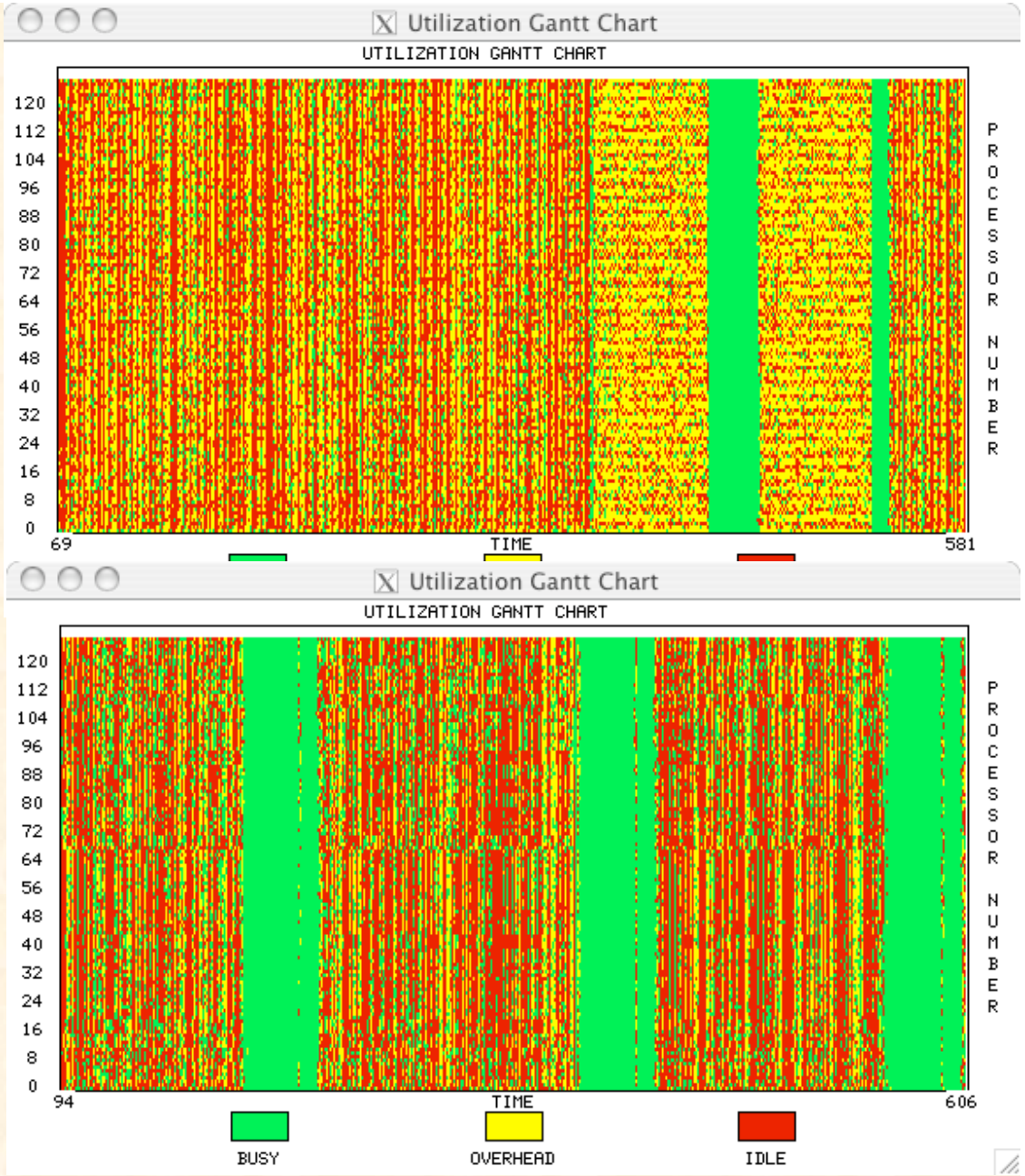
Performance Impact of MPI Tuning

Compute (Busy) / Communicate (Overhead and Idle) Utilization Graph before and after MPI optimization for 128 MSP run.



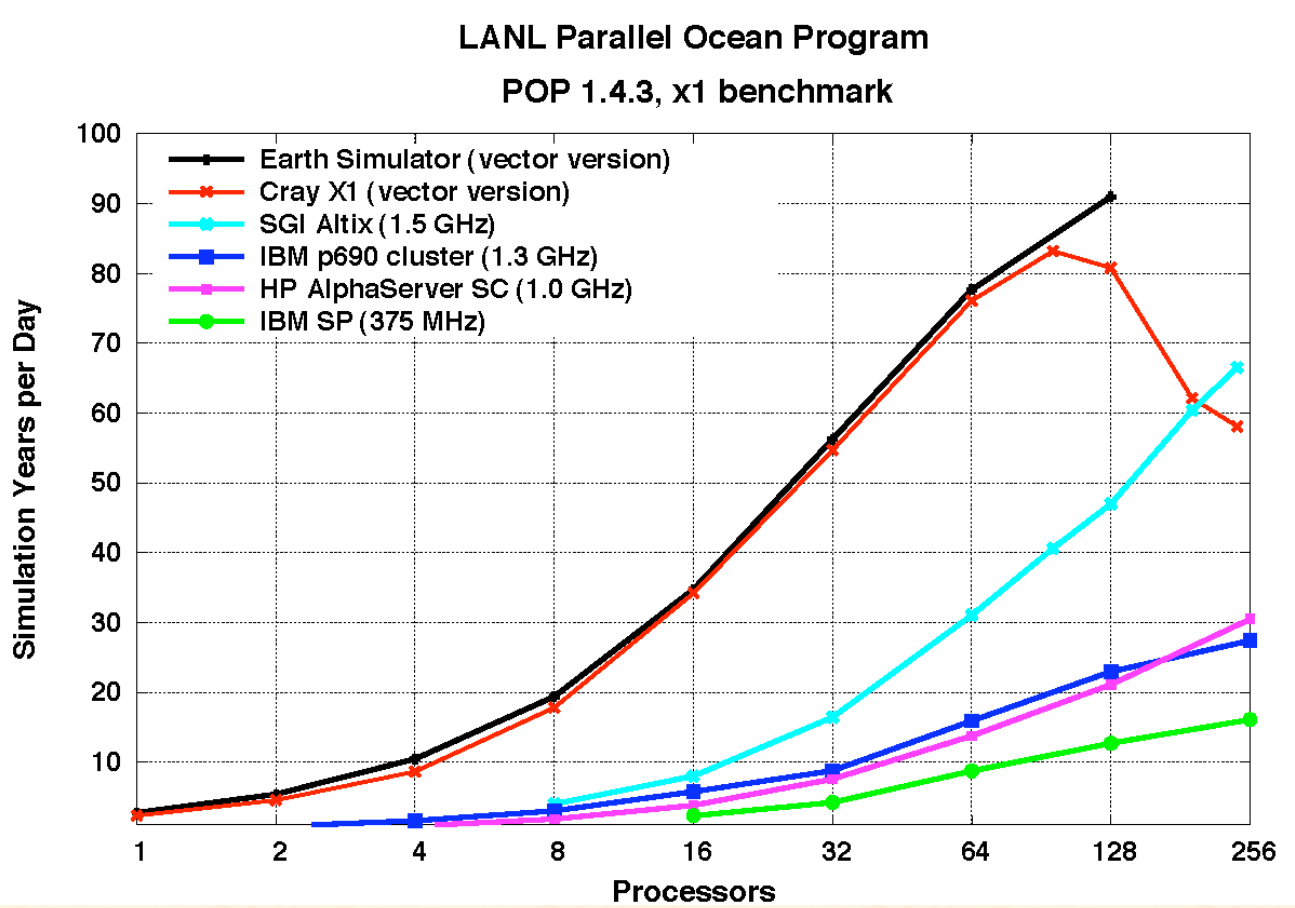
Performance Impact of MPI Tuning

Compute (Busy) / Communicate (Overhead and Idle) Gannt chart before and after MPI optimization for 128 MSP run.



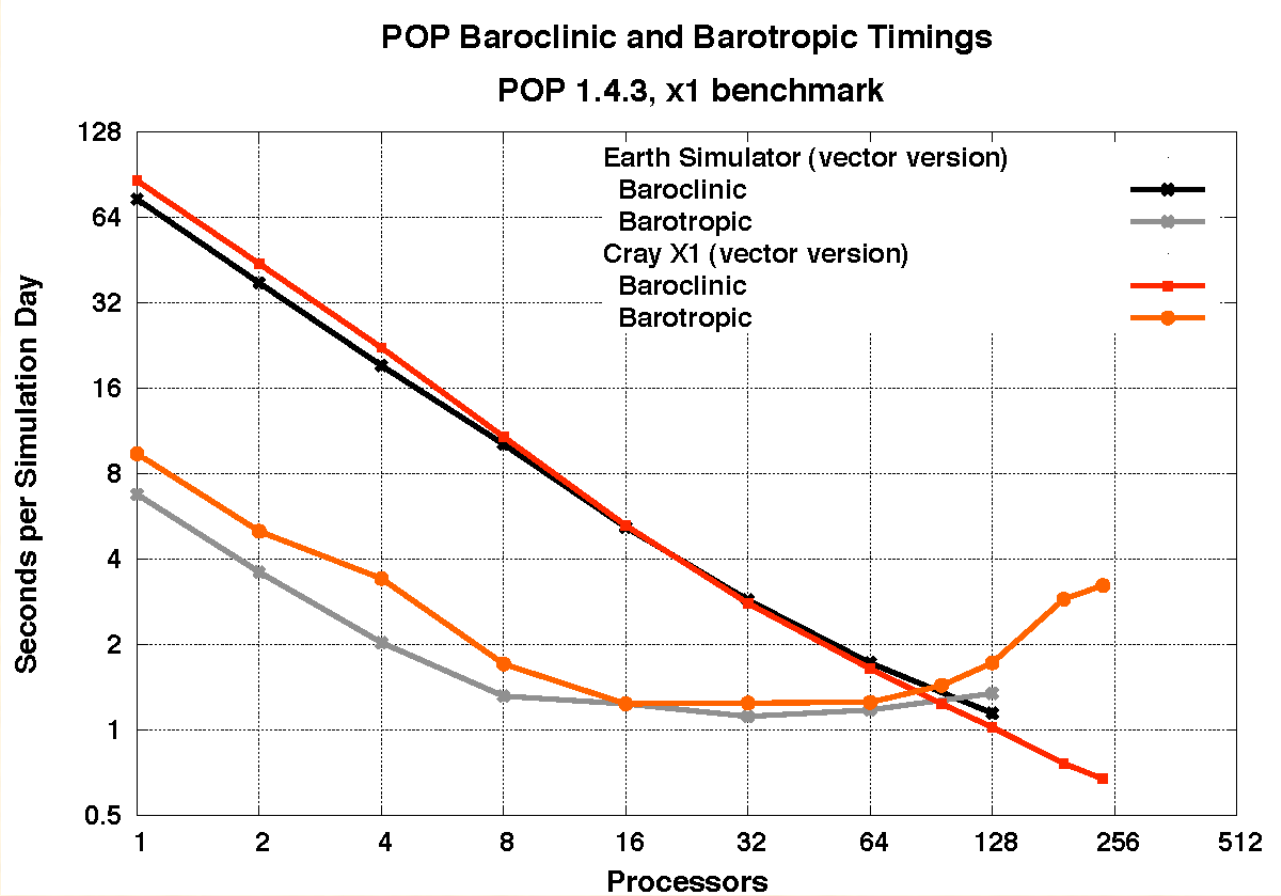
POP Platform Comparison: MPI-Only

Comparing performance and scaling across platforms. Performance on ES40 and X1 are similar when using ES40 optimizations. Performance scales poorly on X1 when using more than 96 processors.



Perf. Diagnosis vs. ES40: MPI-Only

Poor scaling of POP on the X1 is caused by poor scaling in the barotropic phase. The solver in the barotropic phase is dominated by latency-sensitivity communication routines. Lower latency communication options could fix this problem.



POP Co-Array Fortran Optimization

- Replaced MPI implementation with Co-array Fortran for two routines:

  NINEPT_4: Weighted nearest neighbor sum for 9 point stencil, requiring a halo update. Used to compute residuals in conjugate gradient solver in barotropic phase.

  GLOBAL_SUM: Global sum of the “physical domain” of a 2D array. Used to compute inner product in conjugate gradient solver in barotropic phase. MPI version used MPI_Allreduce.

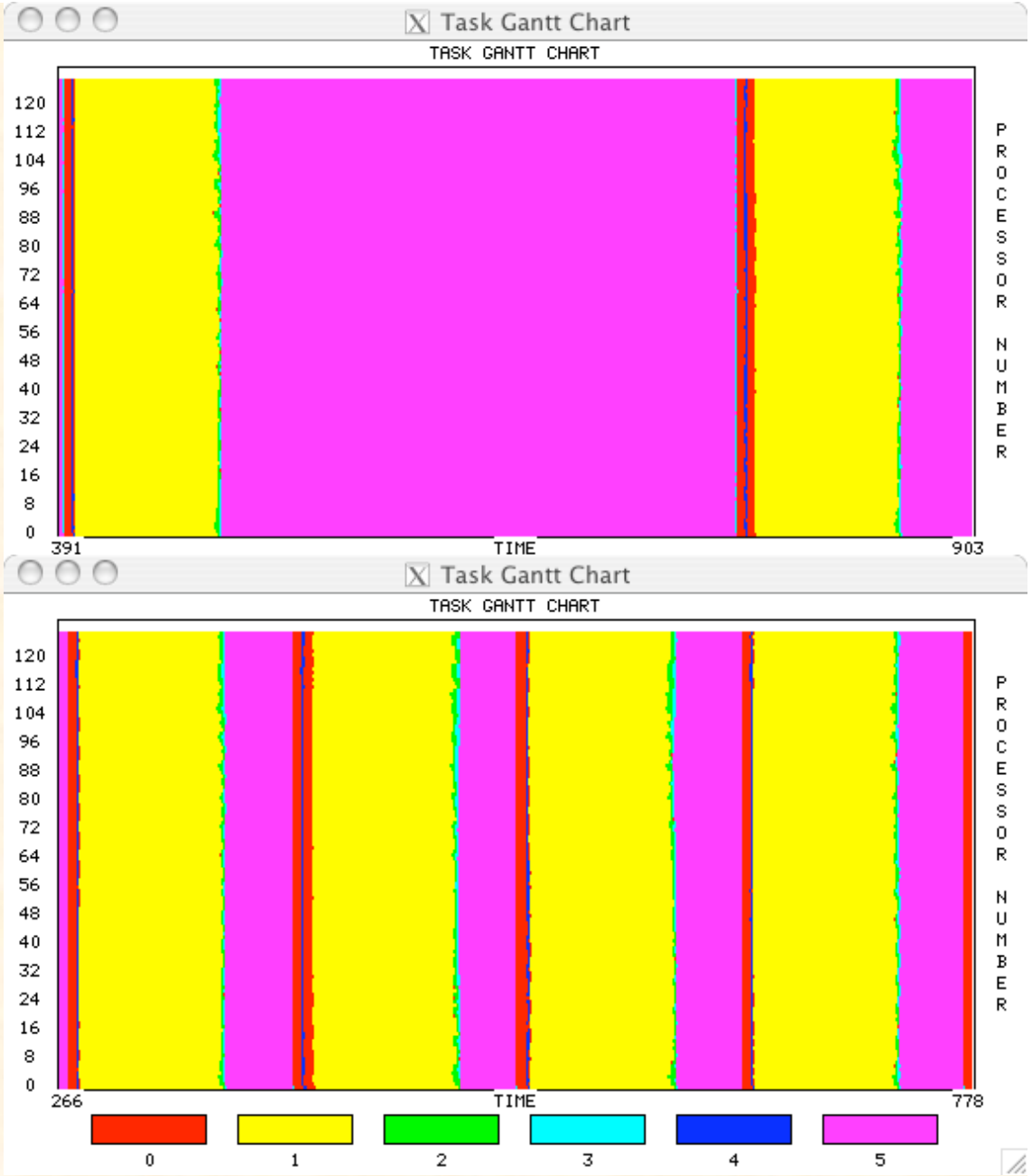
POP Co-Array Fortran Optimization

Simplified Co-Array Fortran version of halo update in NINEPT_4

```
call sync_images()
do n=1,num_ghost_cells
  do j=1,jmt
    XOUT(iphys_e+n,j) = XOUT(iphys_b+n-1,j)[me1p1,me(2)]
    XOUT(       n,j) = XOUT(iphys_e-num_ghost_cells+n,j)[me1m1,me(2)]
  end do
end do
call sync_images()
do n=1,num_ghost_cells
  do i=1,imt
    XOUT(i,jphys_e+n) = XOUT(i,jphys_b+n-1)[me(1),me2p1]
    XOUT(i,       n) = XOUT(i,jphys_e-num_ghost_cells+n)[me(1),me2m1]
  end do
end do
call sync_images()
```

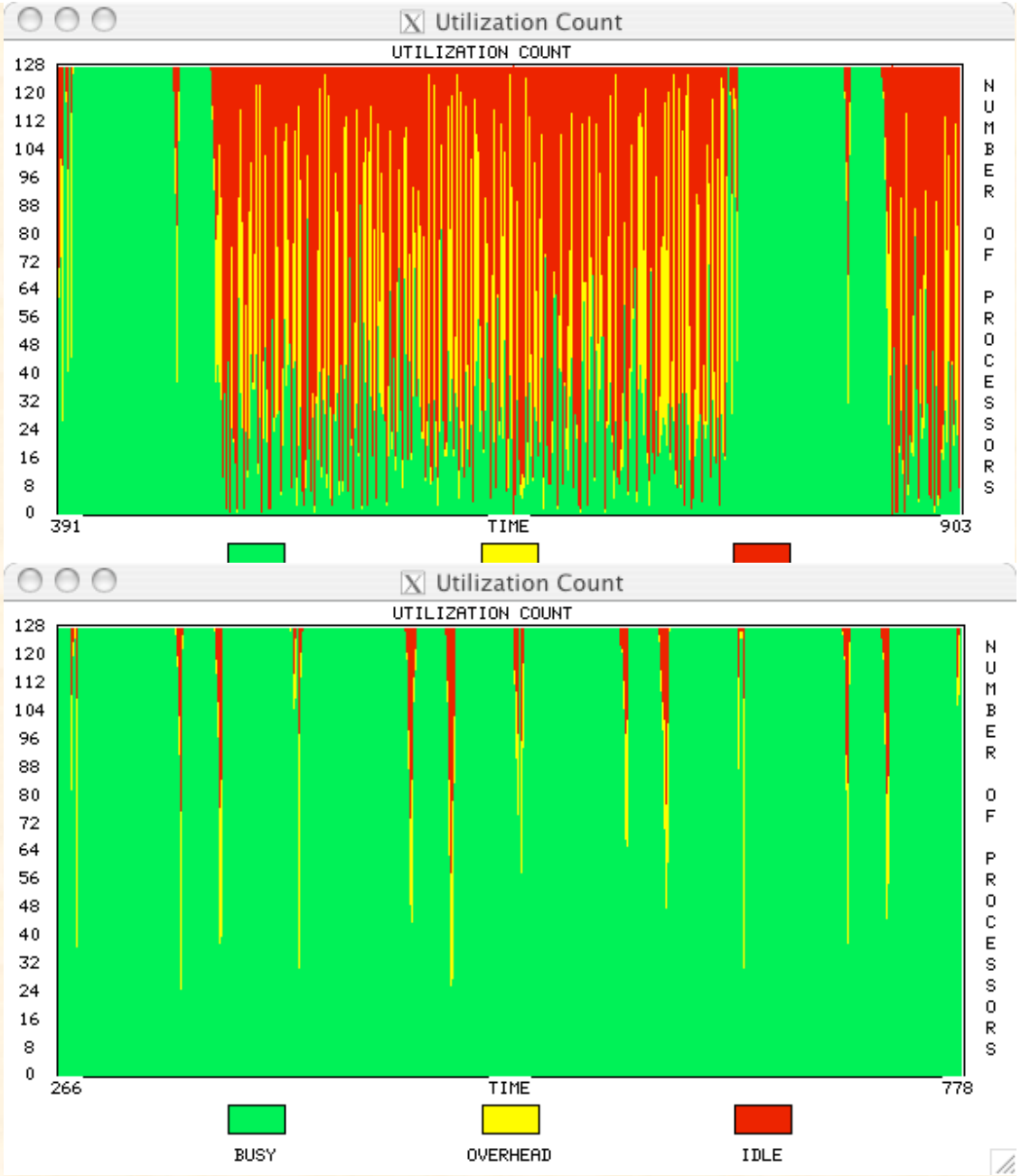
Performance Impact of Co-Array Fortran

Task Gannt chart before and after replacement of MPI allreduce and halo update in barotropic solver (task #5) with Co-Array Fortran for 128 MSP run.



Performance Impact of Co-Array Fortran

Compute (Busy) / Communicate (Overhead and Idle) utilization Graph before and after replacement of MPI allreduce and halo update in Barotropic Solver for 128 MSP run. Time spent in Co-Array Fortran is not marked as Communication overhead. Data indicate that (remaining) MPI overhead is not limiting performance.

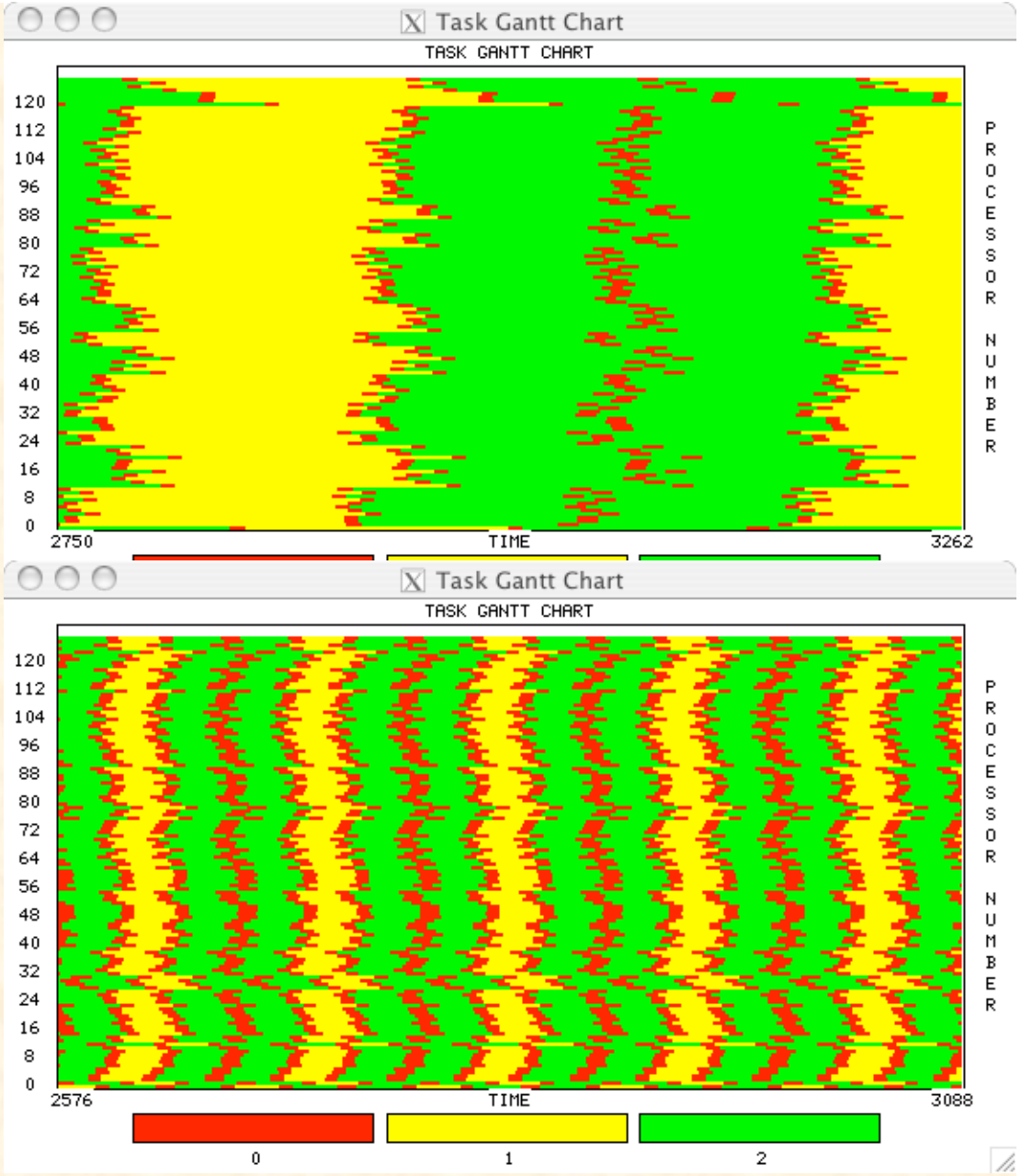


Performance Impact of Co-Array Fortran

Task Gannt chart before and after replacement of MPI allreduce and halo update in barotropic solver with Co-Array Fortran for 128 MSP Run:

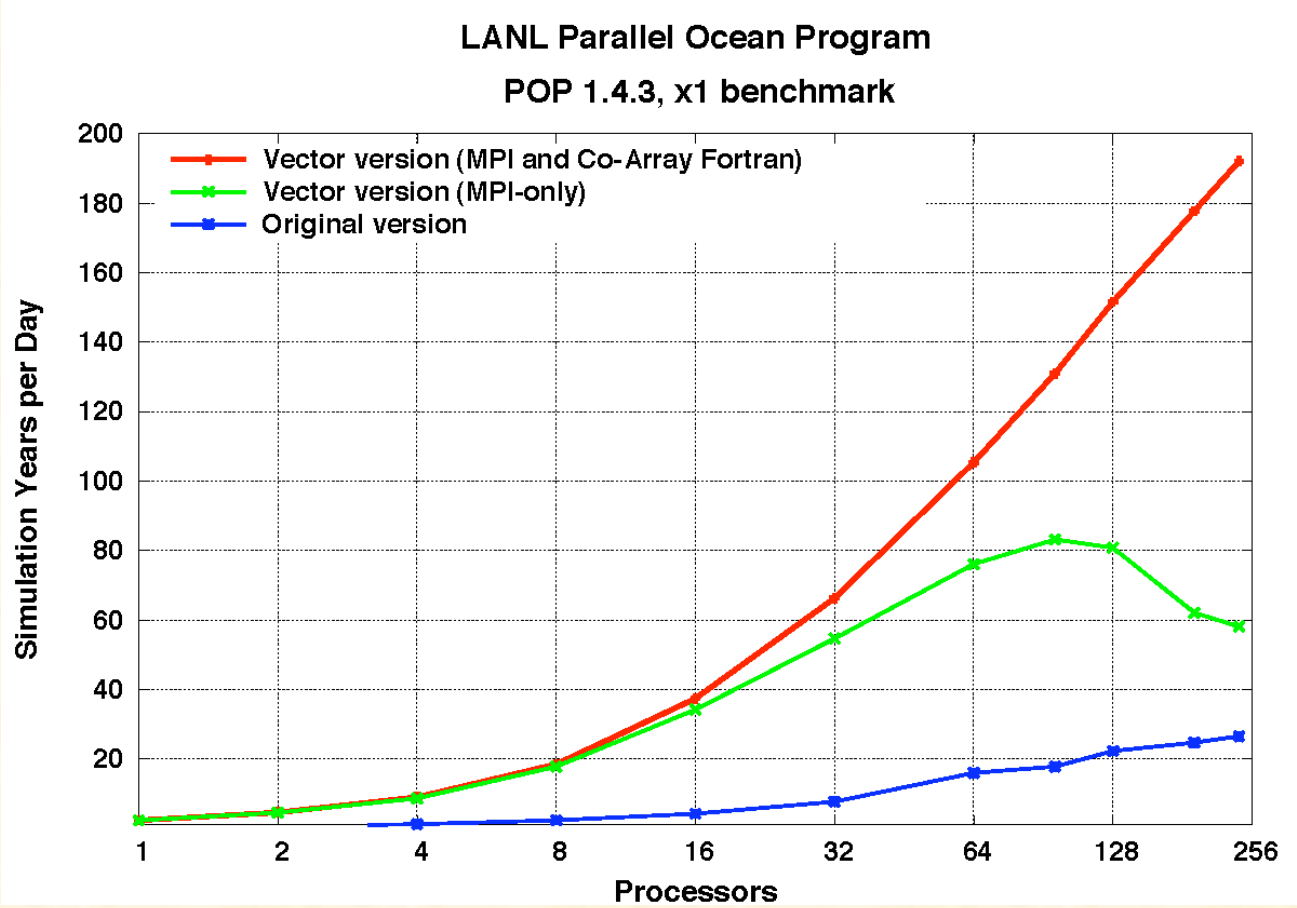
0: “other”
1: Allreduce
2: Halo Update

Data comes from within solver.



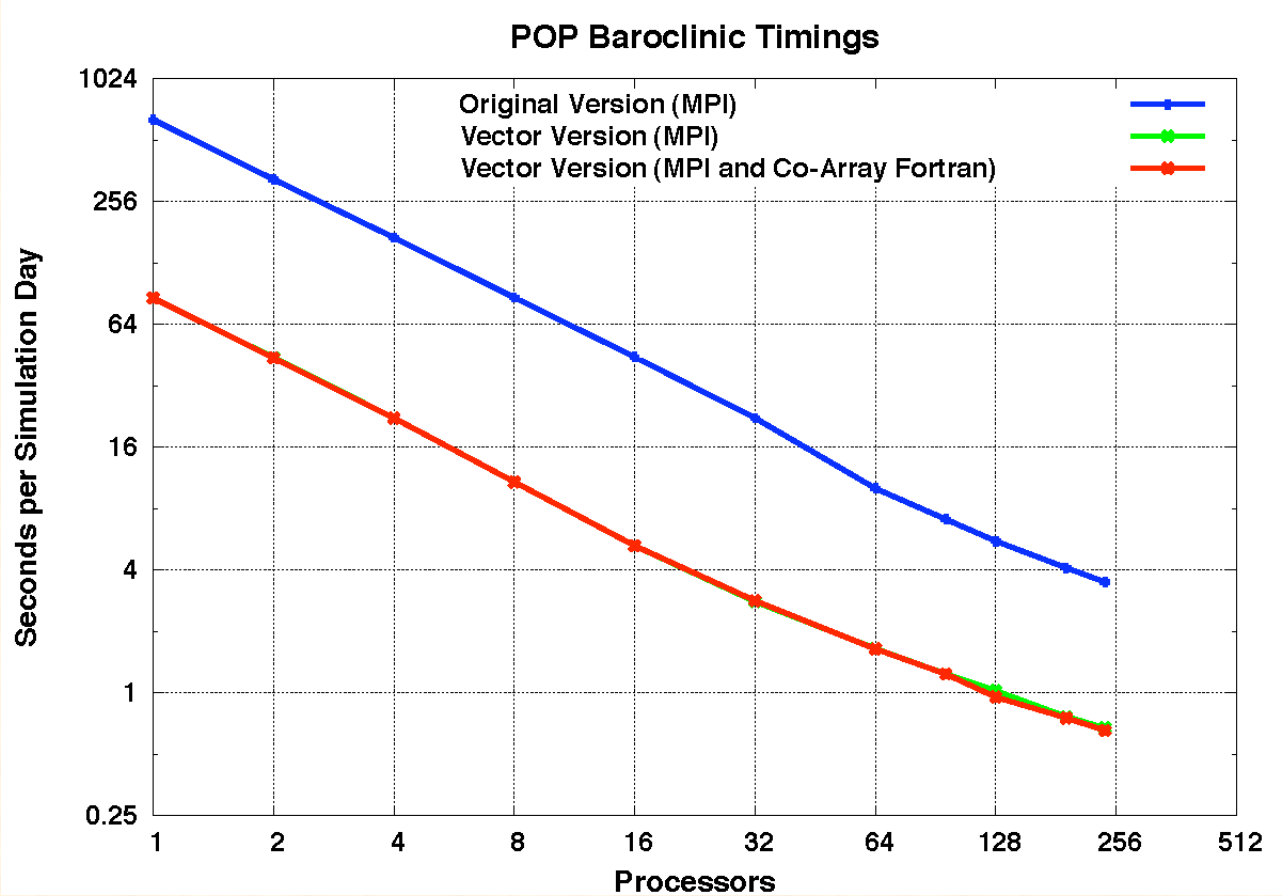
POP Implementation Comparison

Comparing performance of nonvector and vector, and of MPI and hybrid MPI/Co-Array Fortran, implementations.



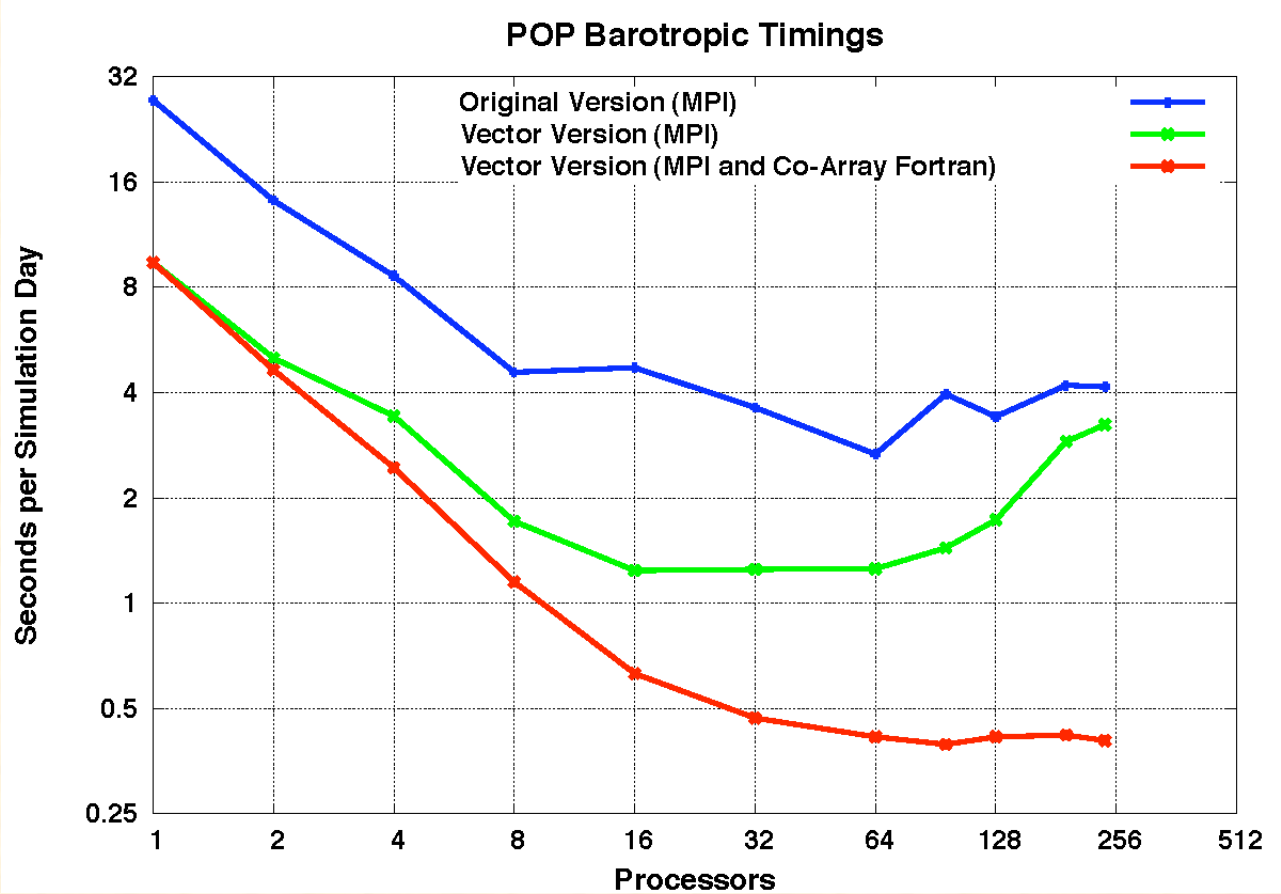
POP Implementation Comparison

Comparing performance of nonvector and vector, and of MPI and hybrid MPI/Co-Array Fortran, implementations for baroclinic phase. The MPI and hybrid vector versions have identical baroclinic performance. The vector version is approx. 8 times faster than the original version for this phase of POP.



POP Implementation Comparison

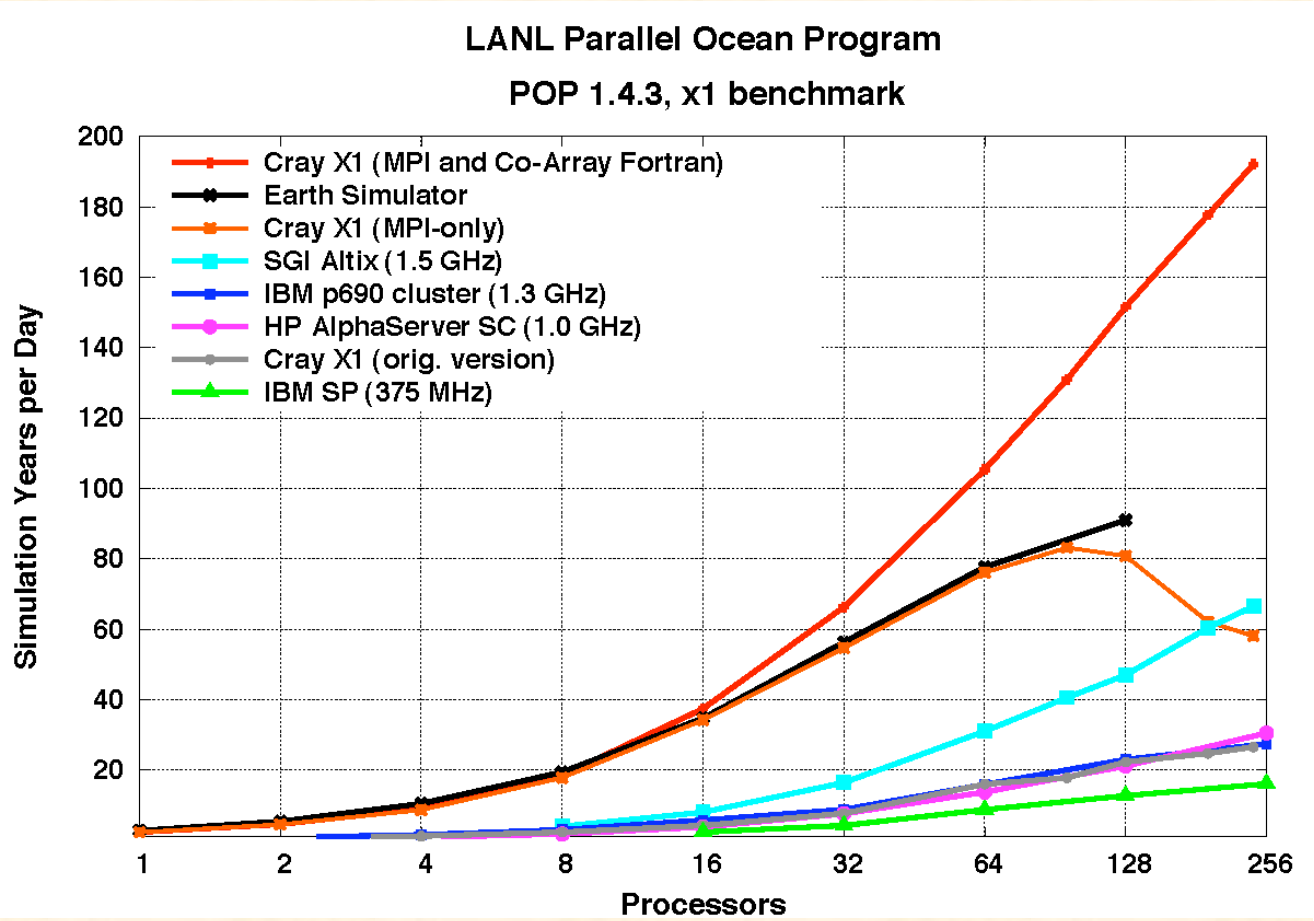
Comparing performance of nonvector and vector, and of MPI and hybrid MPI/Co-Array Fortran, implementations for barotopic phase. The vector and original versions differ in both code structure and MPI protocols. Selective use of Co-Array Fortran extends the scalability of POP significantly.



POP Simulation Rate

Comparing performance and scaling across platforms.

- Earth Simulator results courtesy of Dr. Y. Yoshida of the Central Research Institute of Electric Power Industry
- IBM SP results courtesy of Dr. T. Mohan of Lawrence Berkeley National Laboratory
- X1 at Cray results courtesy of J. Levesque of Cray, Inc. Cray system using beta versions of system software.



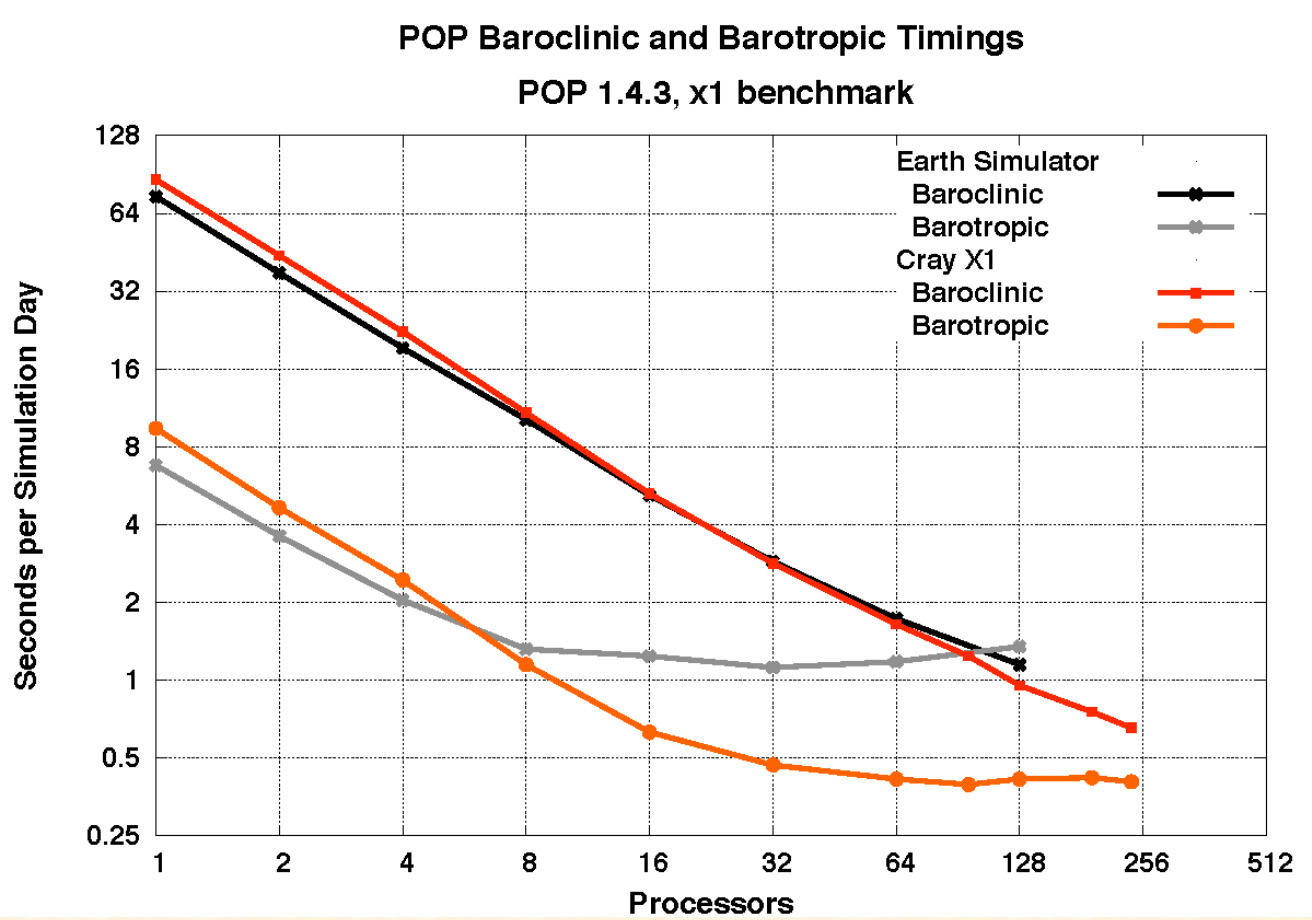
POP Performance Diagnosis vs. ES40

Cray X1

Communication-bound for more than 240 processors, with communication costs just starting to increase.

Earth Simulator

Communication-bound for 128 processors. Better vector performance for large granularity, but worse performance compared to X1 for small granularity (shorter vectors).



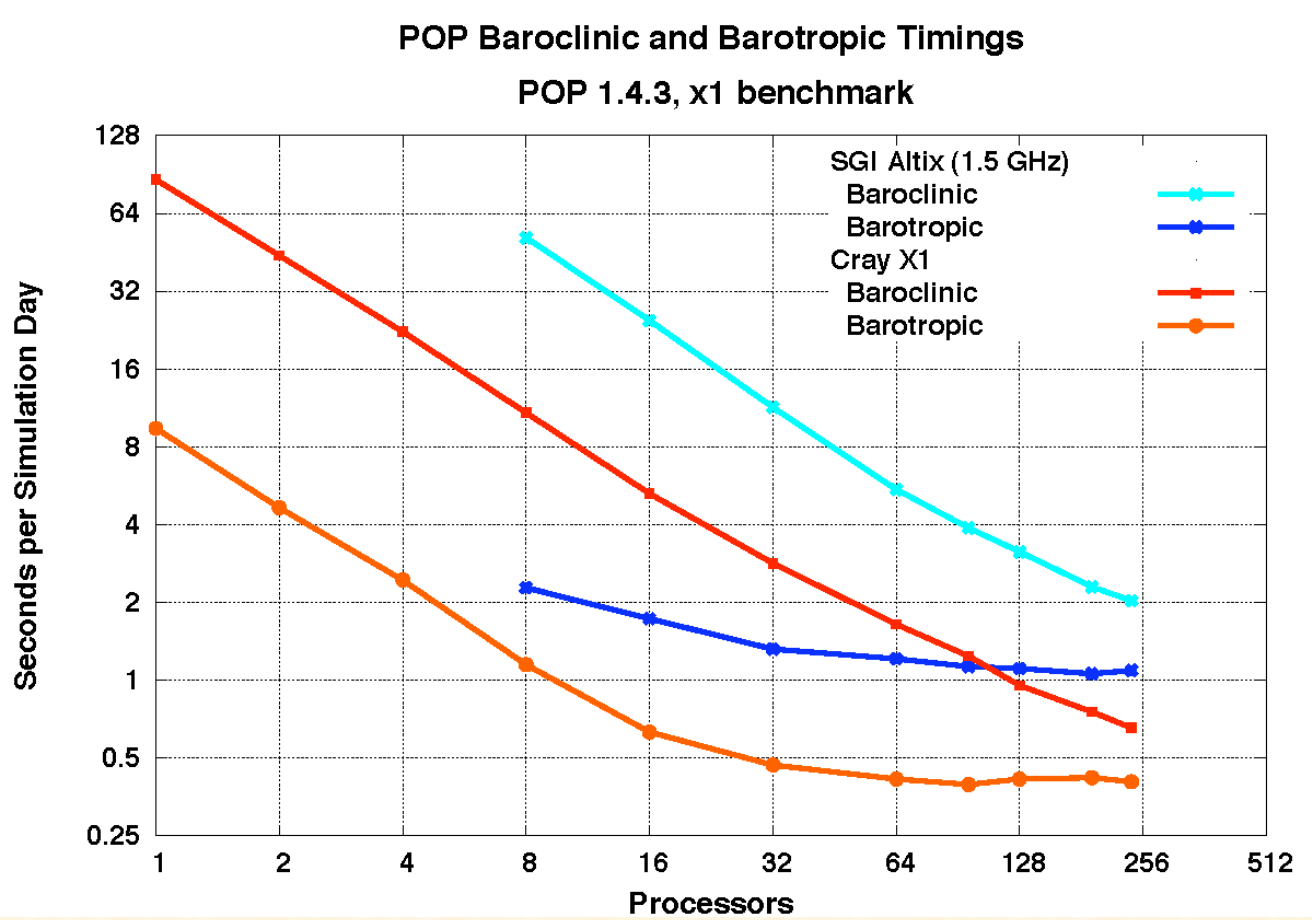
POP Performance Diagnosis vs. Altix

Cray X1

Communication-bound for more than 240 processors, with communication costs just starting to increase.

SGI Altix

Not yet communication bound. Using MPI point-to-point and collectives for barotropic. Initial experiments with SHMEM do not show significant improvement.

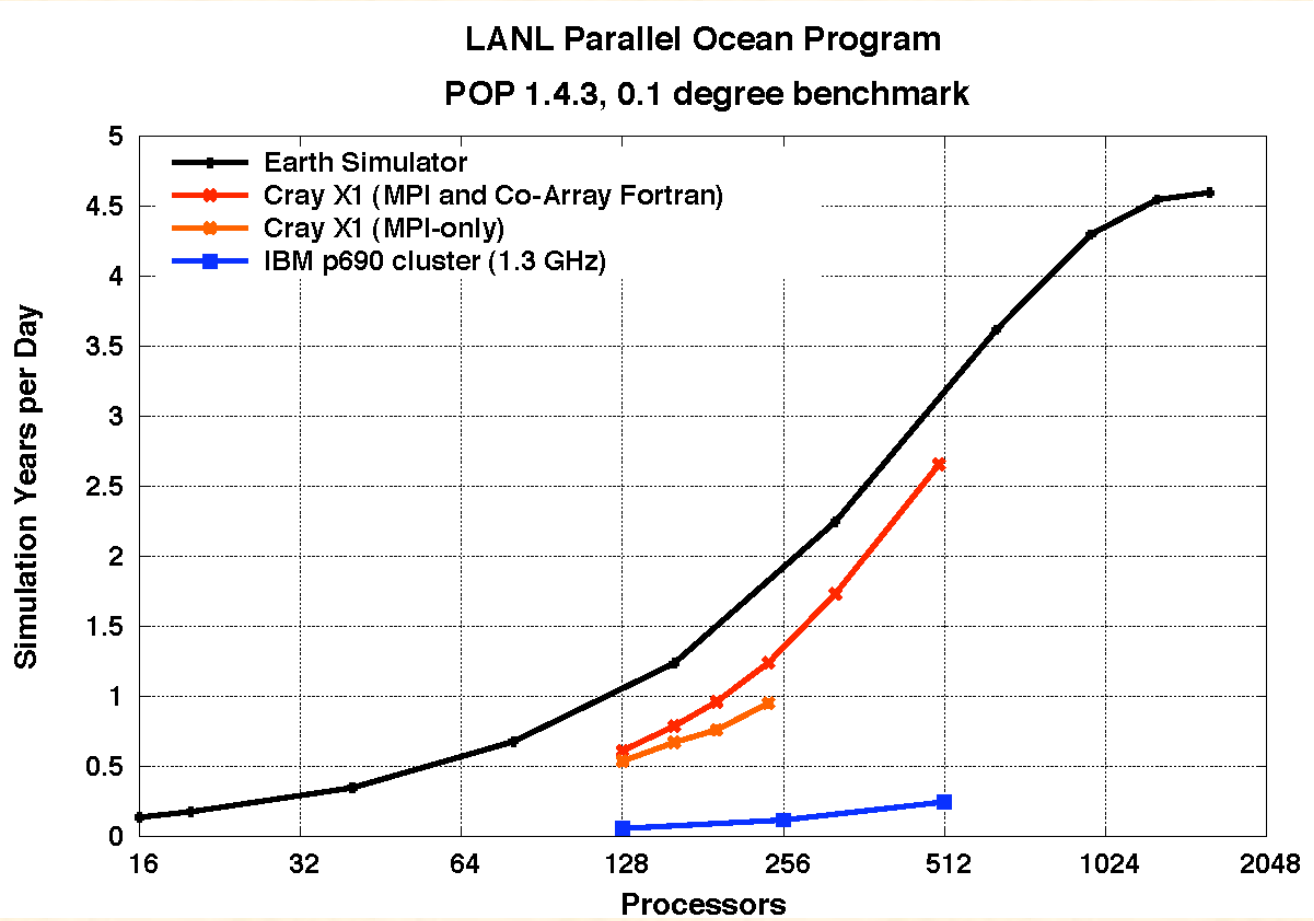


POP Simulation Rate: 0.1 benchmark

Comparing performance and scaling across platforms for a 0.1 degree benchmark problem (3200 x 2400 x 40 grid).

- Earth Simulator results courtesy of Dr. Y. Yoshida
- X1 results collected May 11-15, 2004

X1 performance is good compared to IBM, but lags behind that of the Earth Simulator. Larger problem size changes importance of Cray-specific vector optimizations.



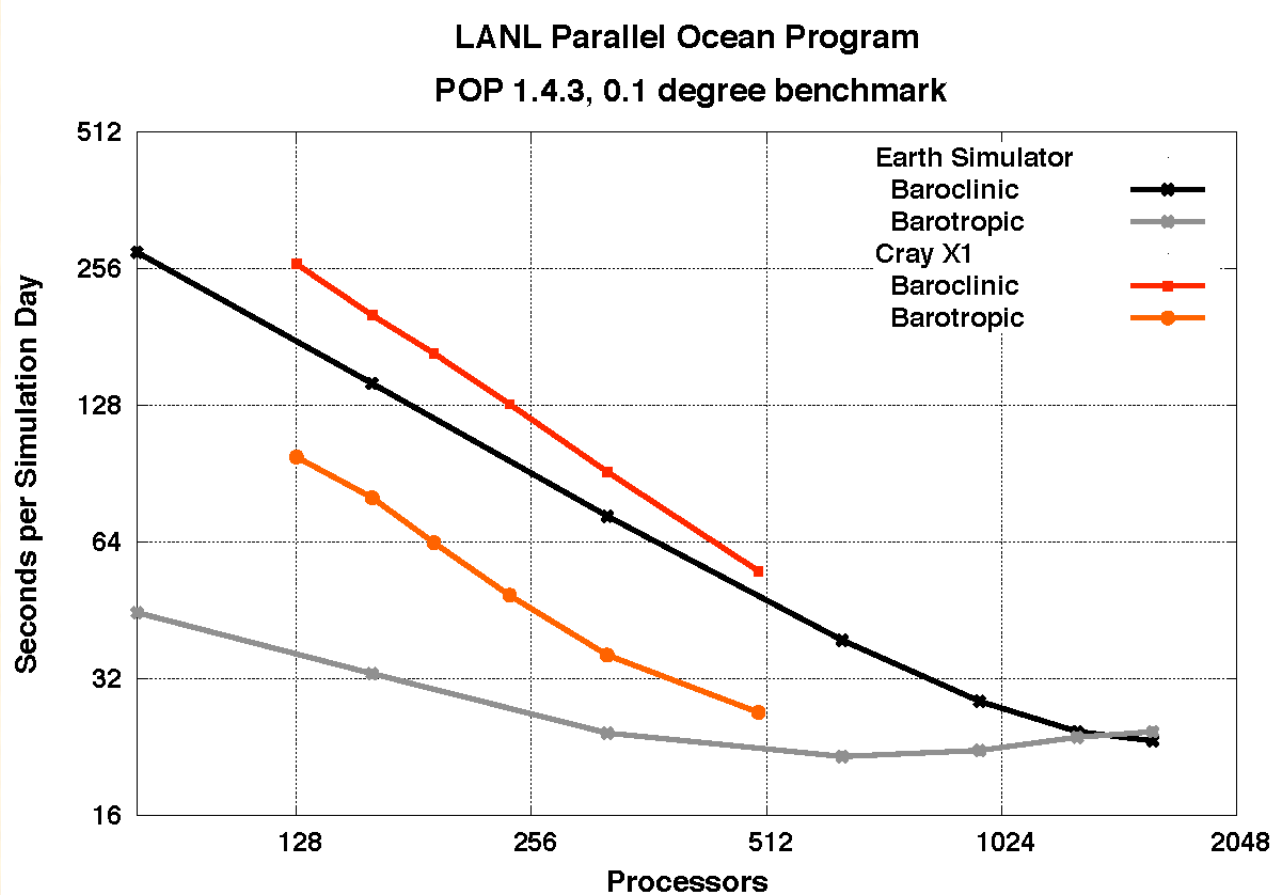
POP Performance Diagnosis vs. ES40

Cray X1

Both baroclinic and barotropic slower than on the Earth Simulator. Scaling is better on the Cray, and performance may be better for 1024 processors. Barotropic is computation-bound up to 500 processors.

Earth Simulator

Communication-bound for 1280 and more processors.



POP Optimization Lessons

- Comparing performance between platforms can lead to insight into how to improve performance.
- If possible, steal / resurrect vector version(s) of codes.
  - NEC optimizations were a good starting point for the Cray X1. Many of the modifications were unnecessary for the X1, but they did no harm.
  - NEC vector optimizations are not sufficient for the large benchmark problem.
- Using MPI derived types can hurt performance (currently).
- Performance of latency-sensitive MPI collective and point-to-point commands limit scalability (currently). Co-Array Fortran can be used selectively to address this problem.
- Performance visualization can be very useful. It is important to design the “performance experiment” carefully in order for visualization to be effective.

PSTSWM Algorithm Comparisons

PSTSWM is a parallel algorithm testbed with many different parallel algorithm options. Part of the process of comparing performance on different platforms is to determine the optimal parallel algorithm settings on a given platform. On the Cray X1, we compared the performance of different parallel algorithms for a 1D decomposition over latitude. To compute the Legendre transform (also over latitude), we examined two options:

- "transpose" the data decomposition to decompose over wavenumber (what was longitude) and use a serial Legendre Transform.
- leave the data decomposition as is and use a distributed algorithm to compute the Legendre transform.

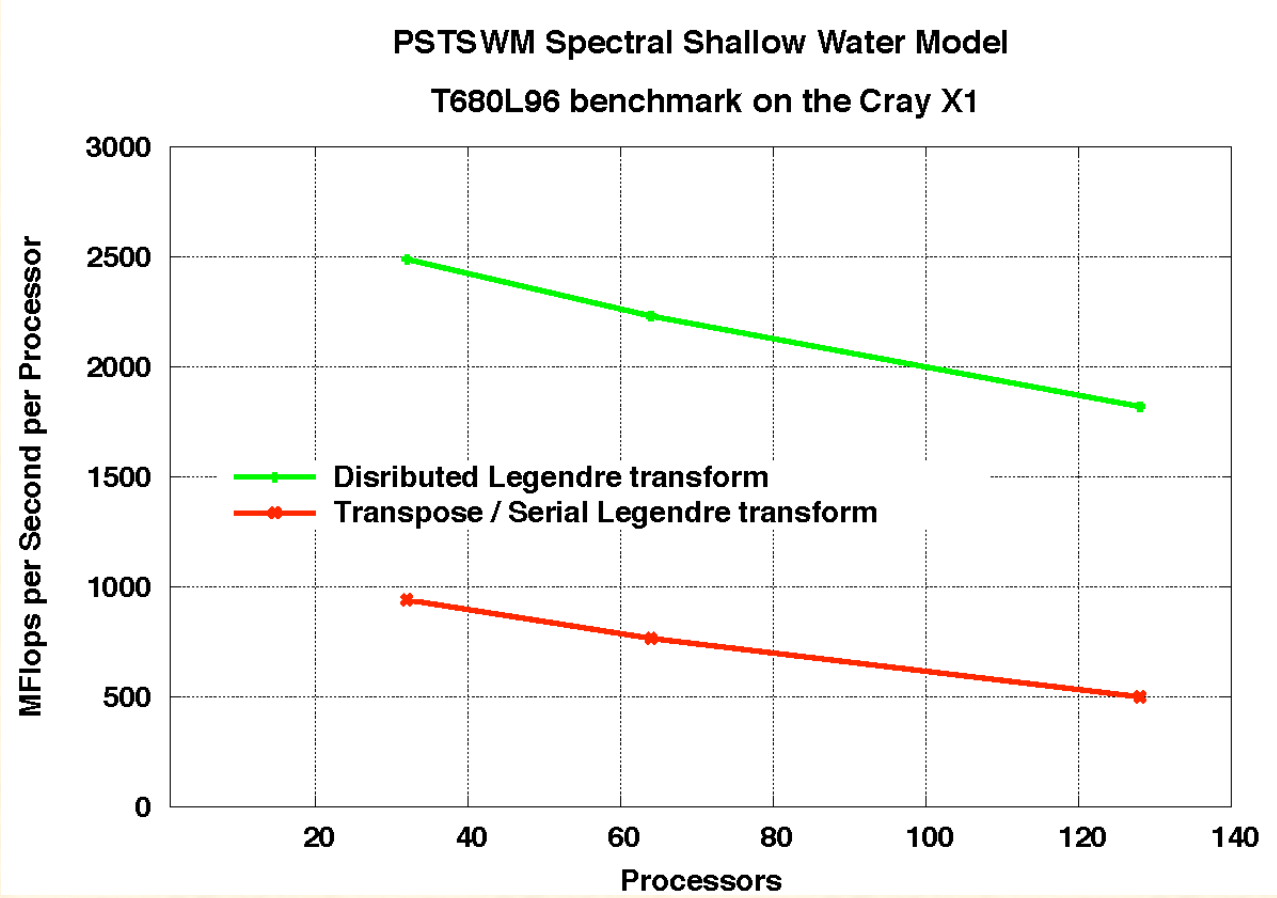
Given the high bandwidth interconnect and the increased vectorization potential in a serial Legendre transform, we expected the transpose approach to be the most efficient.

PSTSWM Algorithm Comparison

Problem Size
T680L96 (1024 x 2048 x 96)

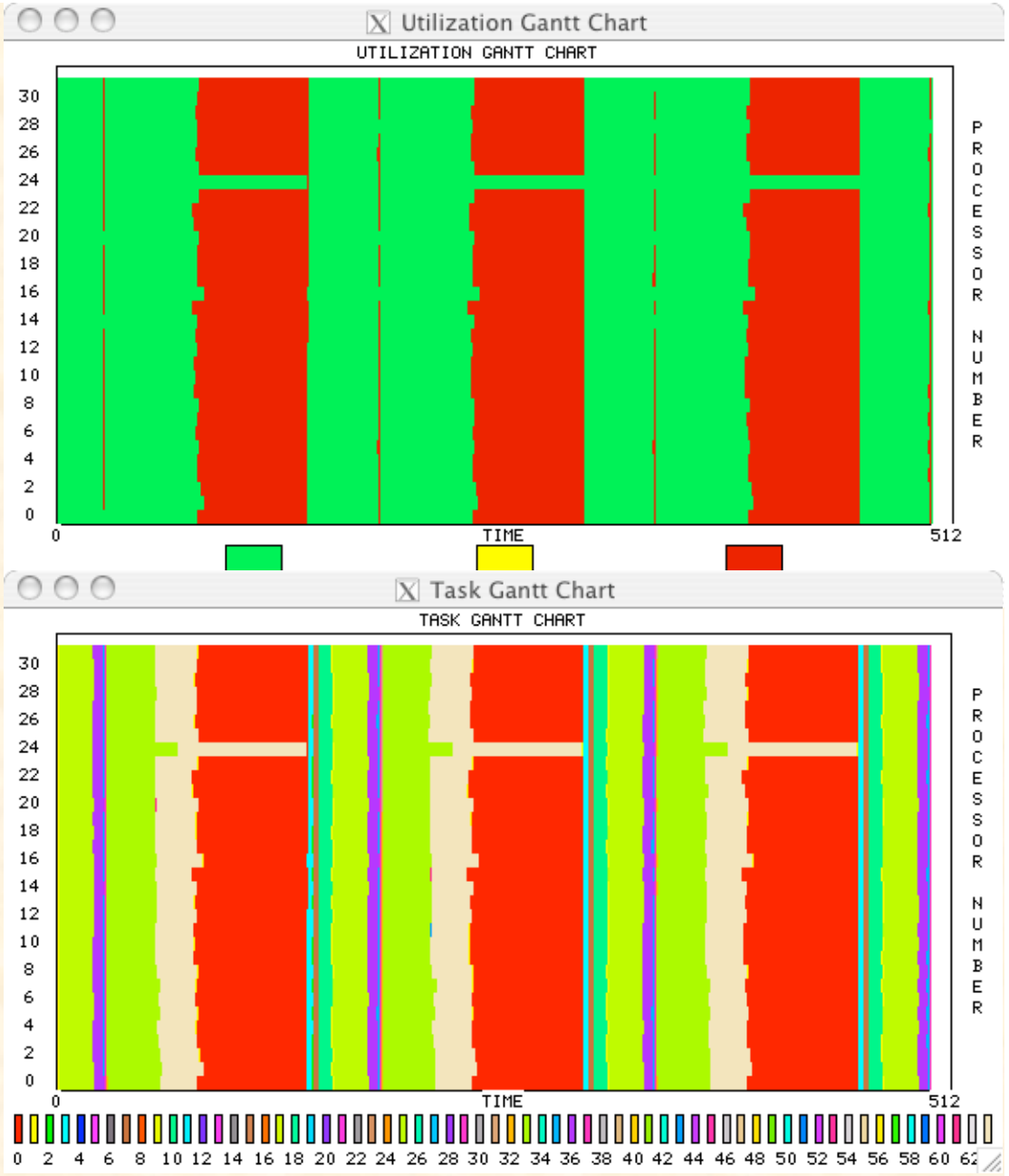
Legendre Transform Algorithms
- distributed algorithm (ring pipeline comm. pattern)
- transpose / serial transform (mpi_alltoallv transpose)

The two algorithms should differ primarily in communication overhead. Loop lengths and cache locality also differ in the transpose and distributed algorithms.



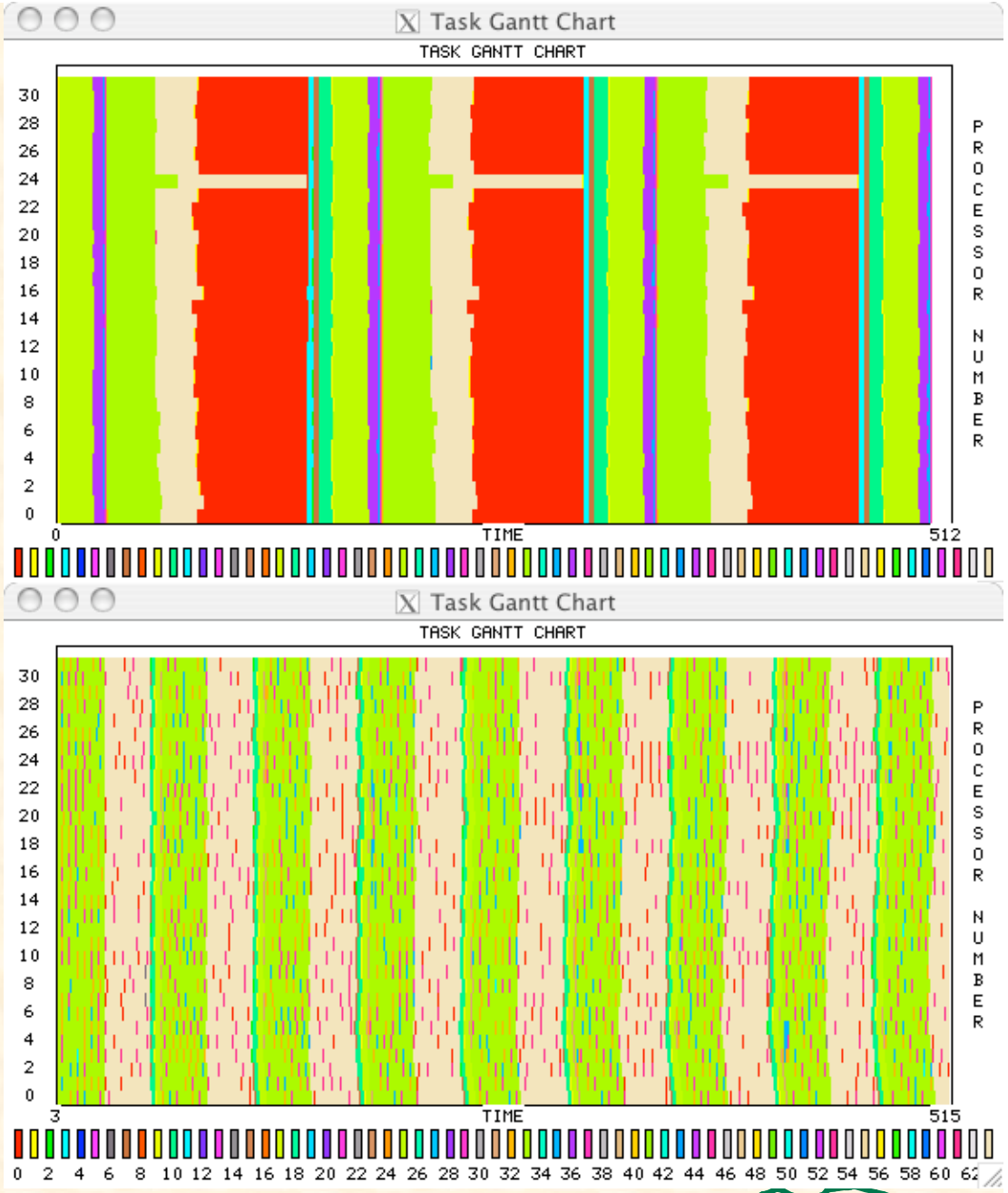
Performance Impact of Load Imbalance

Compute (Busy) / Communicate (Overhead and Idle) and Task Gannt charts for transpose algorithm. The red task is the call to mpi_alltoallv. Process 24 is taking much longer than all of the other processes (for no good reason). The current conjecture is that this is a memory alignment issue.



Performance Impact of Load Imbalance

Compute (Busy) / Communicate (Overhead and Idle) and Task Gannt charts for transpose and distributed algorithms. The distributed algorithm uses multiple communications for the transform rather than one collective as in the transpose. However, it is clear that the distributed algorithm does not suffer from the same load Imbalance issue.

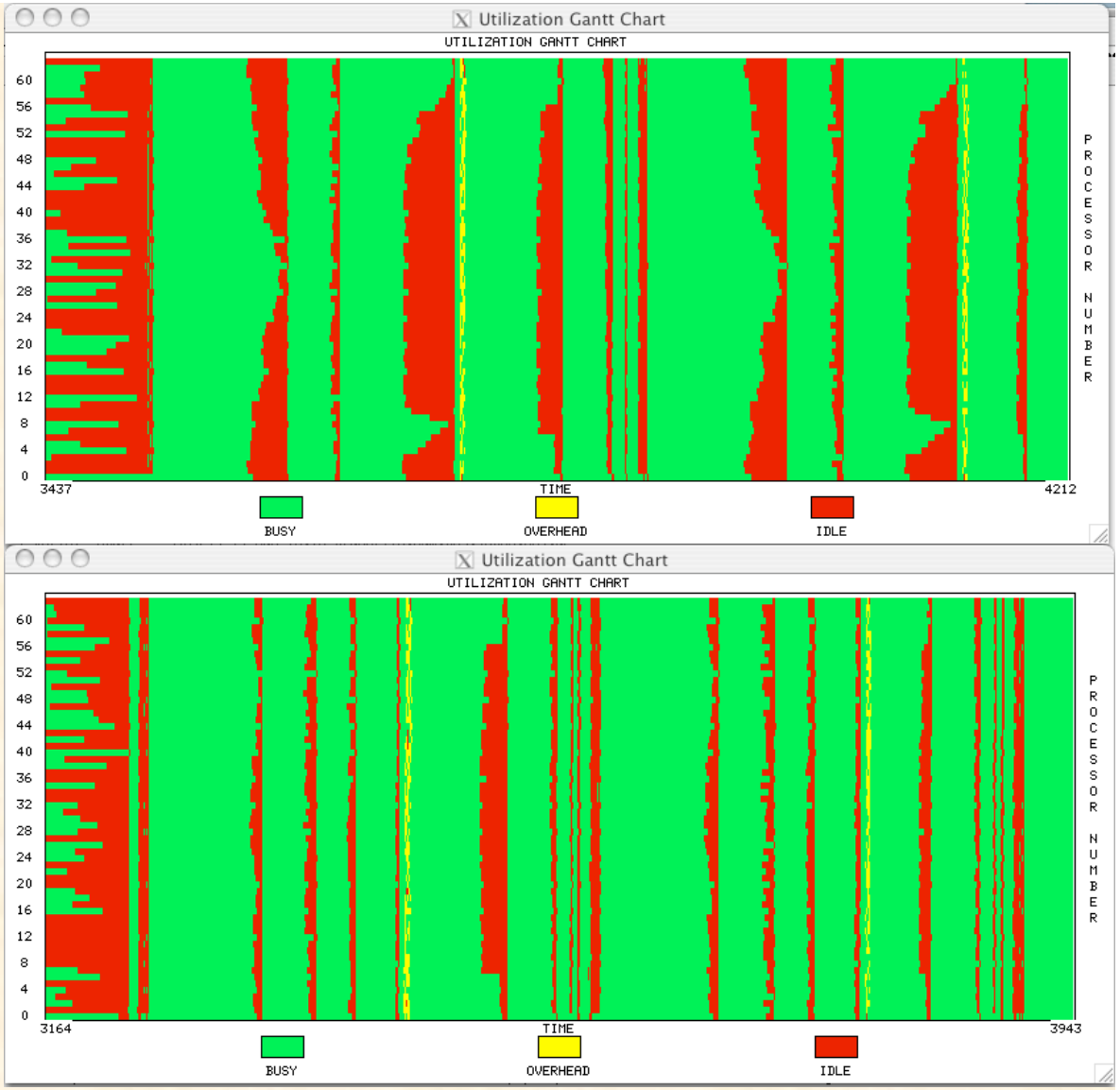


CAM Load Balance

The Community Atmospheric Model (CAM) has known static load imbalances arising from the impact of diurnal and seasonal cycles on the solar radiation calculations. CAM has a number of load balancing algorithms, trading off effectiveness of load balance with communication cost. On most systems, the communication cost of implementing the optimal static load balancing is higher than the performance gain in eliminating the load imbalance. The high bandwidth communications available on the Cray X1 makes it worthwhile examining this in more detail.

Performance Impact of Load Balancing

Compute (Busy) / Communicate (Overhead and Idle) Gannt charts with and without load balancing. Load balancing was (unexpectedly) even more important on the X1 than on other systems. Because of the good interprocessor communication performance, the optimal static load balancing was the best choice.



Load Balancing Optimization Lessons

- Performance expectations can be important (even if ultimately wrong) in determining whether poor performance is a bug or a feature. (Performance models are even better.)
- Memory alignment can impact performance (significantly) on the Cray.
- Load imbalance is “bad”. The Cray communication performance is “good”. Load balancing schemes may be useful on the Cray even if they were not useful on other systems.

EVH1 Vector Lengths

The EVH1 application represents an important kernel in the "TeraScale Simulations of Neutrino-Driven Supernovae and Their Nucleosynthesis" SciDAC project. EVH1 is based on VH1, a multidimensional ideal compressible hydrodynamics code written by John Blondin at NCSU. EVH1 is an MPI code that uses 2nd order operator splitting and 1D Lagrangian hydrodynamics in each coordinate direction (explicit Piecewise Parabolic Method).
In the parallel implementation, the domain decomposition is restructured as each coordinate direction is treated, to keep the given direction on processor. The benchmark problem is a simulation of the Sedov-Taylor blast wave in 2D spherical geometry.

EVH1: sweepx template

```
 6.                  subroutine sweepx
...
25.  1----------<      do k = 1, kmax ; do j = 1, js
…
47.  1            ! Input rho, ei,  ye; return pressure, gammas, temperature.
49.  1                  call eos_result(nmin,nmax)
50.  1            !  Set boundary conditions
51.  1                  call sweepbc( nleftx, nrightx )
52.  1            !  Compute volume elements
53.  1                  call volume ( ngeomx )
56.  1            !  Evolve flow
57.  1                  call ppm    ( ngeomx, ntot, zparax )
69.  1            !  Put updated values into buffer for call to MPI_ALLTOALL
…
80.  1---------->     enddo ; enddo
```

EVH1: riemann template

```
1.                    subroutine riemann ( lmin, lmax, game, gamc, …
…
69. 1-----<           do 27 n = 1, 8
70.  1 MV--<             do 25 l = lmin, lmax
71.  1 MV                   pmold(l) = pmid(l)
72.  1 MV                   gamfl(l)  = 0.5*(gclft(l)+1.0)/gclft(l)
…
86.  1 MV                   pmid (l) = pmid(l) + (umidr(l) - umidl(l)) *
87.  1 MV           &                        (zlft (l) * zrgh (l))/ (zrgh (l) - zlft (l))
87.  1 MV                   pmid (l) = max(smallp,pmid(l))
88.  1 MV--> 25          continue
…
96.  1----->    27  continue
```

EVH1 Vector Length

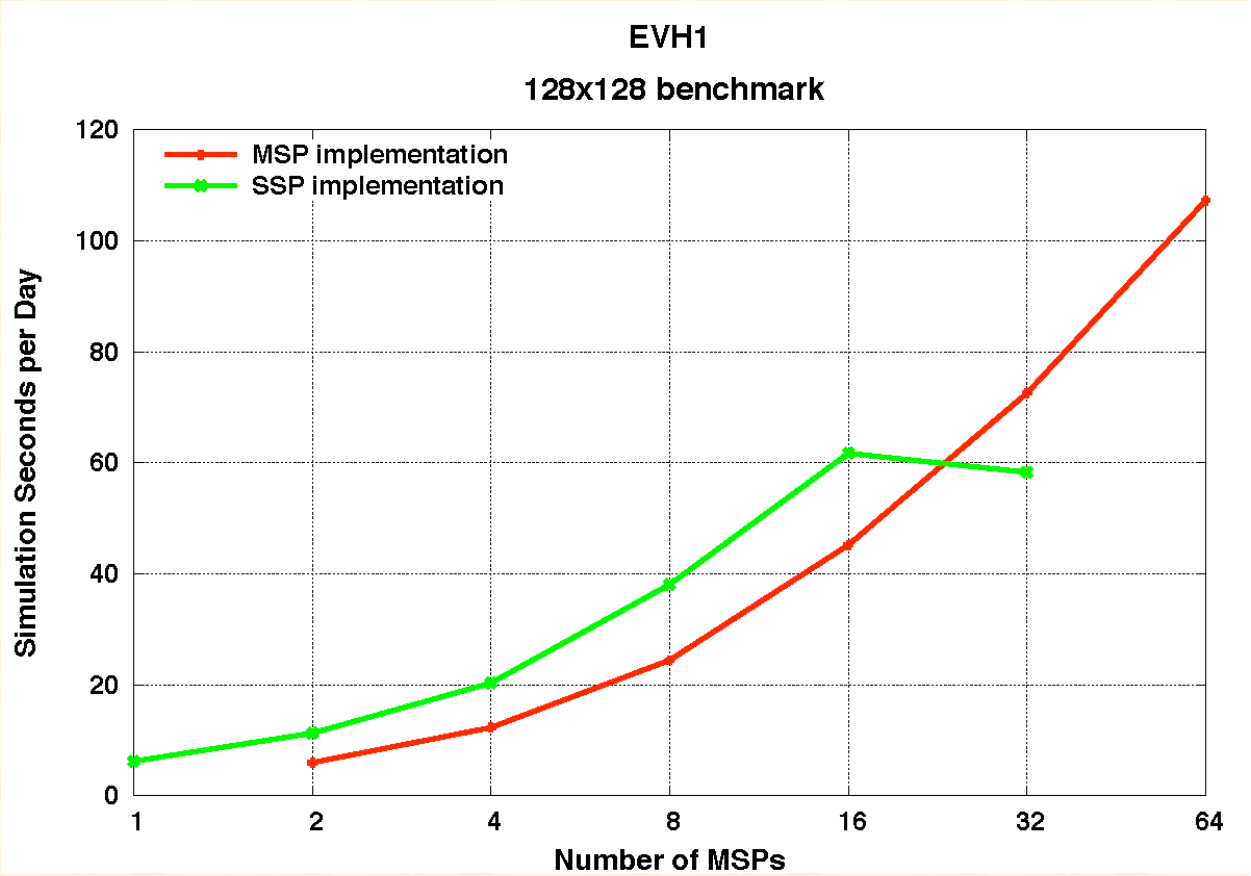
One dimensional temporaries are defined in modules and used to carry intermediate results between subroutines. In consequence, the only loops that can be streamed or vectorized are the 1D inner loops in routines such as riemann. The performance of three benchmark problem sizes were examined:

- 128x128
- 256x256
- 512x512

The max vector length for these three problems is 128, 256, and 512, respectively. In MSP mode, the 128x128 problem can only use half of the vector HW. In SSP mode, assigning an MPI process to each SSP, the target vector length is only 64, so all of the vector HW is used. Additionally, the scalar work between loops is parallelized over 4 times as many (SSP) scalar units in SSP mode. However, the MPI overhead increases as a function of the number of MPI processes.

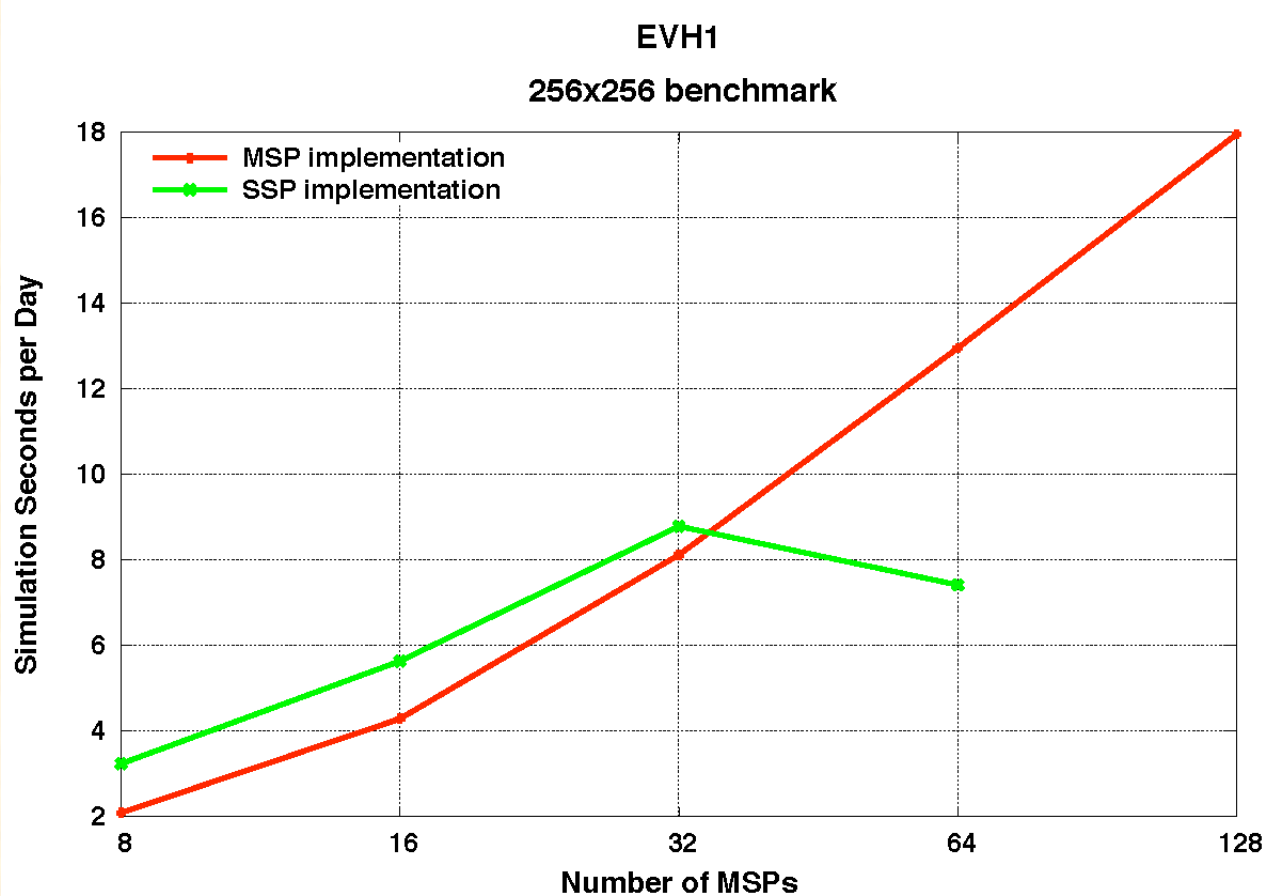
EVH1: MSP vs. SSP

A loop length of 128 is only half what is needed by the MSPs, and SSP performance is approx. twice that of MSP for small process counts. For large process counts, communication overhead begins to dominate, and SSP performance falls below that of MSP. For the 1D decomposition, the maximum number of processes is 128, which is more limiting for SSP mode than for MSP mode.



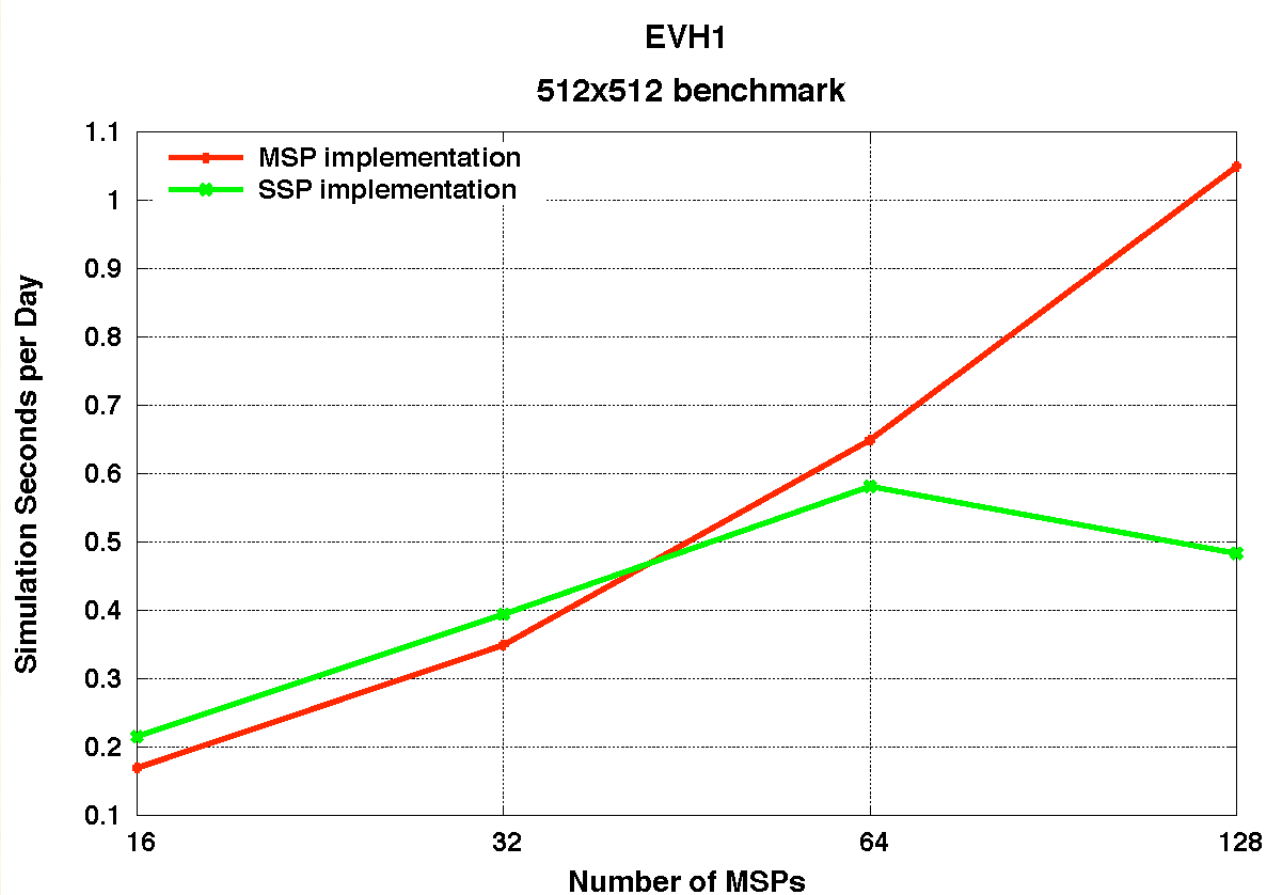
EVH1: MSP vs. SSP

A loop length of 256 should be sufficient to keep the vector HW busy, but the subroutine overhead and work between loops still gives an advantage to the SSP mode for small process counts. Again, at large MSP counts, the higher communication overhead of using SSPs (and more MPI processes) eliminates its advantage in computational rate.



EVH1: MSP vs. SSP

A loop length of 512 is plenty long enough to keep the vector hardware busy. But the serial code and subroutine overhead still gives an advantage to the SSP mode for small MSP counts.



Vector Length Optimization Lessons

- For EVH1, streaming is restricted to the same (inner) loops that were vectorized, and the target vector length differs for SSP and MSP mode.
- SSP mode allows additional MPI parallelism, which can speed up scalar work that can not be streamed in MSP mode.
- If vector length is long enough, MSP mode allows all vector HW to be used with fewer MPI processes, which can decrease MPI communication (and communication cost).
- Defining 1D temporaries in modules and using them to share intermediate results between subroutines prevents vectorization and streaming on outer loops, limiting performance on the Cray. Eliminating this and moving outer loops into subroutines, or enabling more aggressive inlining, *should* improve MSP mode performance compared to SSP mode.

Questions ? Comments ?

For further information on these and other evaluation studies, visit

http://www.csm.ornl.gov/evaluation .