

A Hybrid MPI-OpenMP Implementation of an
Implicit Finite-Element Code on
Distributed Shared Memory Architectures

G. Mahinthakumar, North Carolina State University
Faisal Saied, National Center for Supercomputing Applications

April 18, 2000

Outline

- Introduction
 - Goal
 - MPI, OpenMP, and Hybrid modes
- Our Implementation
- Performance Results
- Conclusions

Goal

- Study performance implications of MPI, OpenMP, and Hybrid Models on different parallel architectures using an implicit FEM code
- Our OpenMP implementation is very straightforward
 - Our target is only for moderate number of CPUs

MPI vs OpenMP

- MPI (Message Passing Interface)
 - Distributed memory programming model
 - Required for distributed memory parallelization
 - Most implementations also work on shared memory
 - Vendor optimized and free versions available
 - Supported for most scientific computing languages
 - Fortran, C/C++, Java (?)
- OpenMP
 - Shared memory thread programming model
 - Alternative to standard unix thread programming (e.g. pthreads)

Advantages of OpenMP

- Unified standard for thread level parallelism (portability)
- Easy to get things going fast on shared memory architectures
 - Automatic parallelization option gives reasonable performance for some compilers (e.g. O2K)
- Flexibility of incremental bottom-up parallelism
 - We can start by adding directives to compute intensive loops
- Avoids overhead of message passing within a shared memory

Disadvantages of OpenMP

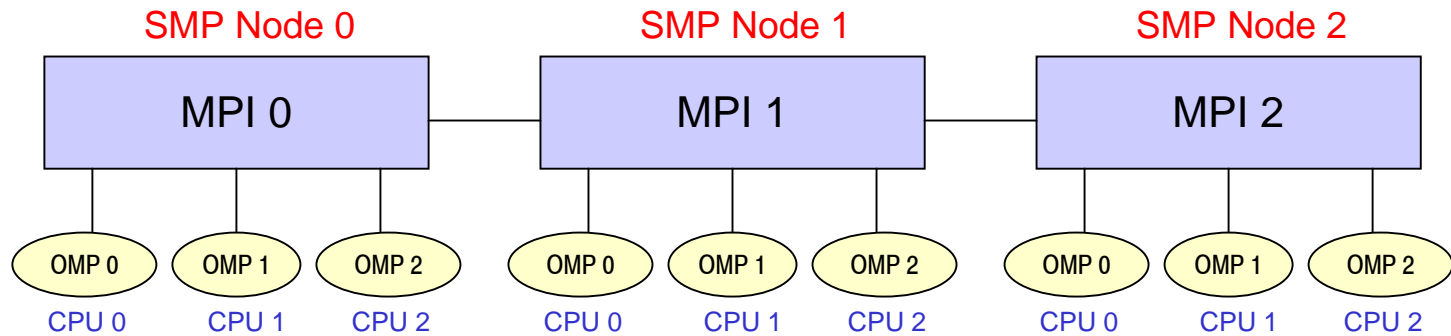
- Reasonable for small number of threads but scalability to large number of threads requires almost as much work or more work as MPI parallelism
 - Still lags behind MPI for scalability
- Difficult for general and performance portability
 - Not all vendor implementations are equal
 - Some features not implemented by some vendors
- Works only within shared memory nodes
 - MPI is still required for going outside nodes

Different Levels OpenMP Implementation

- Automatic compiler parallelization (-apo, -qsmp=auto)
 - No work involved simply recompile code !
 - Not available on all architectures
 - Some vendor implementations better than others
- Loop level parallelization
 - Insert \$OMP PARALLEL DO directives for compute intensive loops
 - Gradually increase parallelism to most loops
- Full fledged approach
 - Decompose memory just like MPI
 - Use private arrays for each thread except the overlapping regions which are shared buffer arrays
 - use THREADPRIVATE directive and COPYIN clause to create the private arrays from common blocks or structures
 - Entire code is a parallel region, all threads are active all the time

Why Hybrid Mode ?

- Hybrid model is an excellent match for the dominant trend in parallel architectures which are made of clusters of multi-cpu shared memory or SMP (Symmetric Multi-Processor) nodes.



Hybrid MPI-OpenMP Programming Paradigm

- Multiple OpenMP threads under each MPI process
 - OpenMP threads can be used within each shared memory node and MPI can be used to communicate across nodes
 - Eliminates message passing within a single shared memory node
 - Nested parallelism is possible in a hybrid model
- Looks like the right approach for DSM architectures comprising of a large number of shared memory SMP nodes each with a moderate number of CPUs
 - Is it really so ? We'll see...

Simple Hybrid Code Example

ORIGINAL MPI CODE

```

=====
! pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
=====
program main
include 'mpif.h'
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
!function to integrate
f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
call MPI_BCAST(n,1,MPI_INTEGER,0,
& MPI_COMM_WORLD,ierr)
h = 1.0d0/n
sum = 0.0d0
do 20 i = myid+1, n, numprocs
x = h * (dble(i) - 0.5d0)
sum = sum + f(x)
20 enddo
mypi = h * sum
! collect all the partial sums
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,
& MPI_SUM,0,MPI_COMM_WORLD,ierr)
call MPI_FINALIZE(ierr)
stop
end
=====

```

HYBRID MPI-OpenMP CODE

```

=====
! pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
=====
program main
include 'mpif.h'
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
!function to integrate
f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
call MPI_BCAST(n,1,MPI_INTEGER,0,
& MPI_COMM_WORLD,ierr)
h = 1.0d0/n
sum = 0.0d0
!$OMP PARALLEL DO REDUCTION (+:sum) PRIVATE (x)
do 20 i = myid+1, n, numprocs
x = h * (dble(i) - 0.5d0)
sum = sum + f(x)
20 enddo
!$OMP END PARALLEL DO
mypi = h * sum
! collect all the partial sums
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,
& MPI_SUM,0,MPI_COMM_WORLD,ierr)
call MPI_FINALIZE(ierr)
stop
end
=====

```

Finite Element Groundwater Transport Code

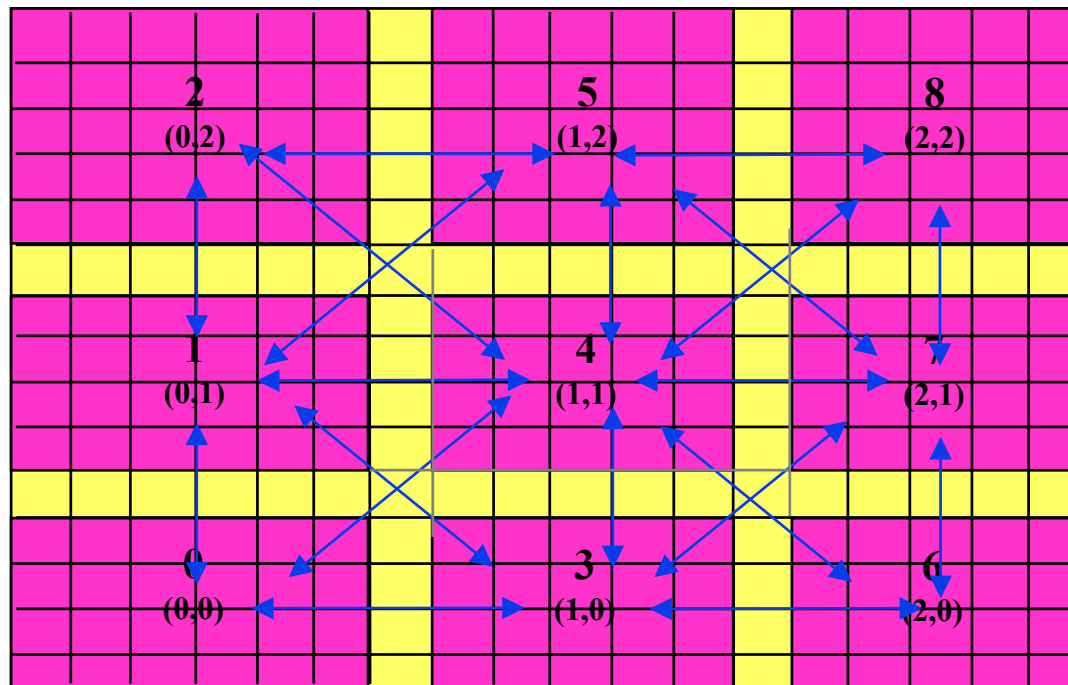
- Multicomponent reactive transport system
 - Advection-dispersion-reaction partial differential equation system
- 3D Hexahedral Elements
 - Logically rectangular grid but irregular geometries supported by distorted elements
- Sparse iterative solvers
 - BiCGSTAB, GMRES, CGS, ORTHOMIN
 - Diagonal and Block Jacobi Preconditioners
- Nonlinearity handled by a sequential iterative algorithm
- Originally parallelized for distributed memory architectures using MPI
 - Scalability tested on many architectures: Cray T3E, IBM SP, SGI O2K, Intel Paragon, Convex Exemplar
 - Primary Language: Fortran (some F90 features used, some C)

MPI Parallelization of FEM Codes

- Domain Decomposition (our approach)
 - Decompose FEM mesh into distinct domains equal to the number of processors
 - Load balancing and communication cost needs to be taken into account
 - Straightforward for structured grids
- Element by element (alternate approach)
 - Do not assemble global matrix
 - Use iterative solvers involving sparse matrix-vector products
 - Perform sparse matrix-vector products element by element
 - Assign each processor a set of elements
 - Good strategy for unstructured grids and higher order elements

Our MPI Implementation

2D Domain Decomposition



↔ interprocessor communication path

□ overlapping processor cells

■ individual processor cells

Our OpenMP Implementation

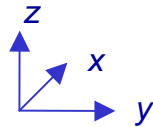
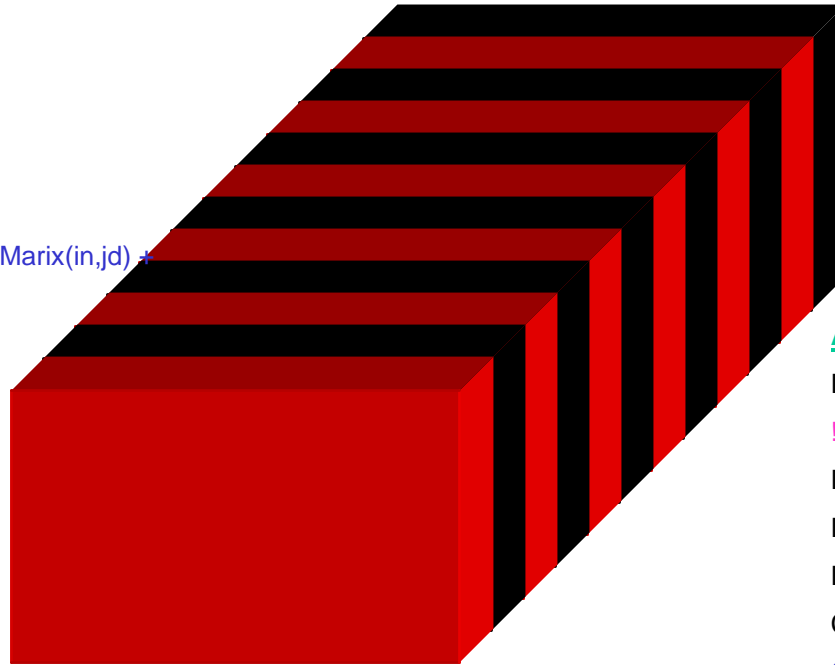
- Insert \$OMP PARALLEL DO directives for most of the do loops in the code
 - This is the strategy we adopted for most of our loops
- Global matrix assembly loop required some modification
 - To avoid thread conflicts during update of global matrix a red-black type scheme was used for the outermost assembly loop
 - Assemble odd numbered y-z planes in the first sweep, and the even numbered y-z planes in the second sweep

OpenMP Parallelization of Global Matrix Assembly Loop

BEFORE

```

Do l=1, nelemx
Do j=1, nelemy
Do k=1, nelempz
Compute element_matrix(i,j,k)
Global_Matrix(in,id) = Global_Marix(in,jd) +
Element_Matrix(i,j,k)
Enddo
Enddo
Enddo
    
```

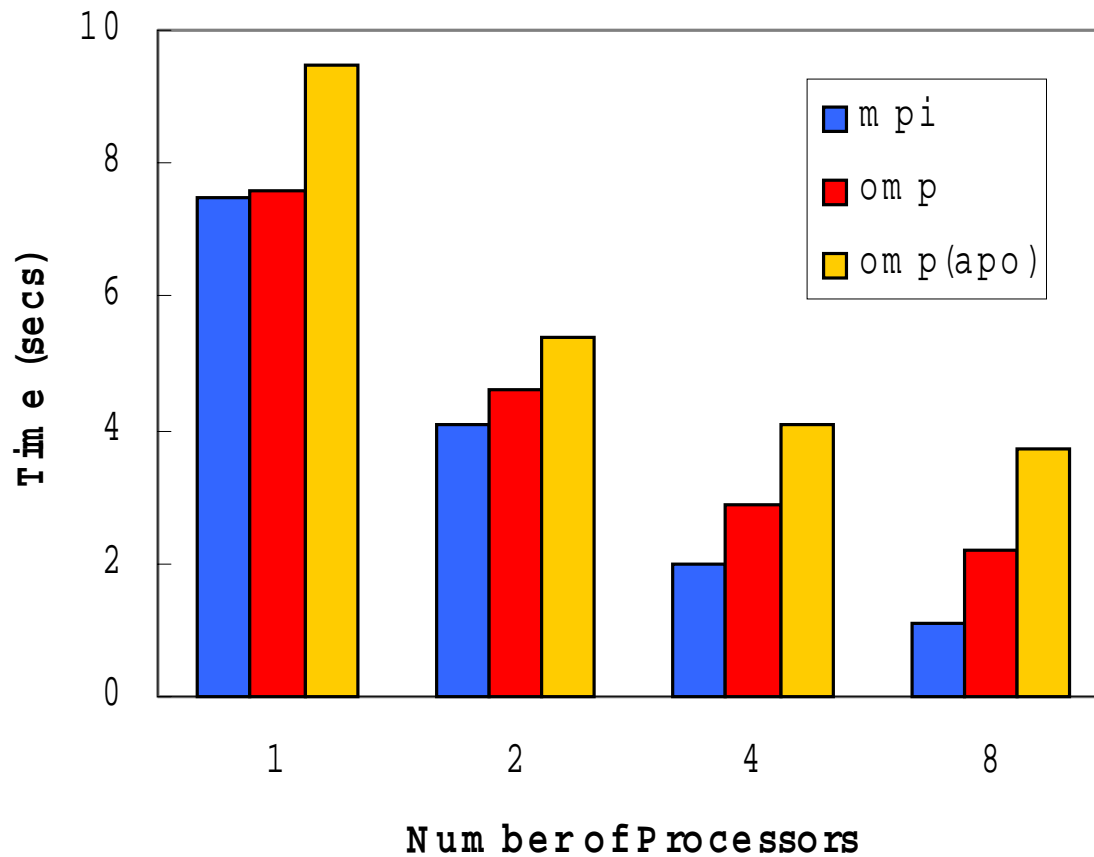


AFTER

```

Do m=1,2
$OMP PARALLEL DO
Do l=m, nelemx,2
Do j=1, nelemy
Do k=1, nelempz
Compute element_matrix(i,j,k)
Global_Matrix(in,jd) = Global_Marix(in,jd) +
Element_Matrix(i,j,k)
Enddo
Enddo
Enddo
Enddo
    
```

Matrix Assembly Times (O2K)



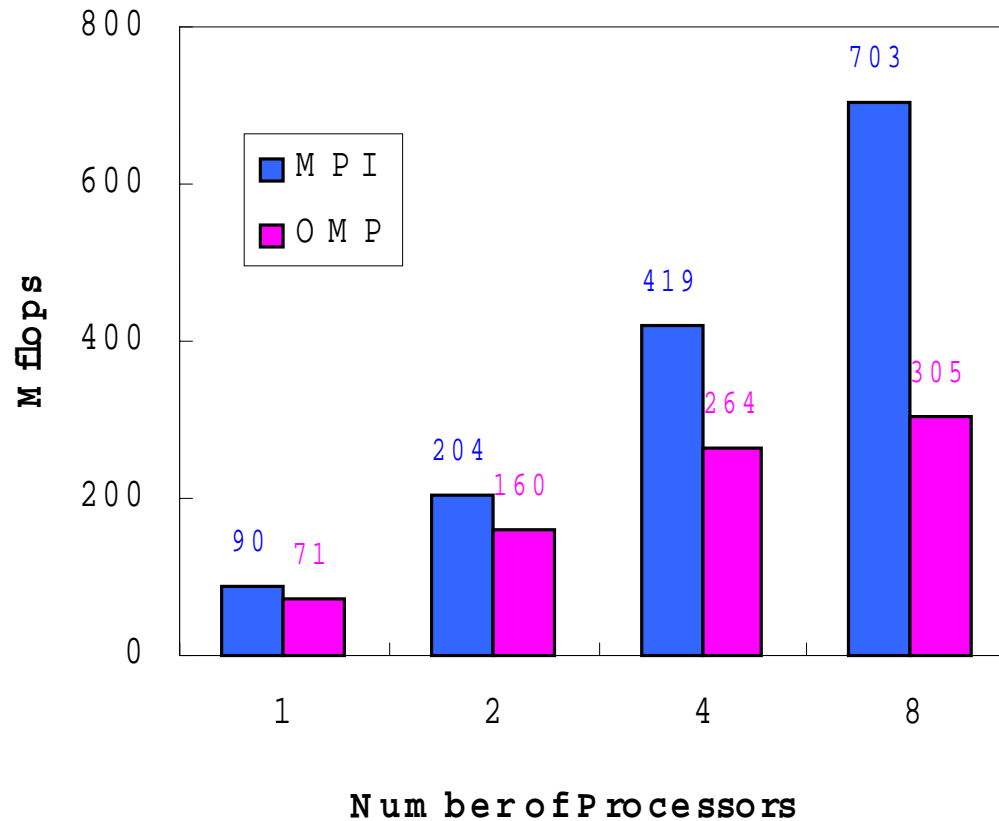
Our Target Architectures

- SGI Origin 2000 at NCSA
 - Non-uniform shared memory (NUMA)
 - Uniform memory access across 2 CPUS
 - 256 CPU and 128 CPU systems
- Small Compaq at ORNL – Limited testing
 - Distributed Shared Memory (DSM)
 - 4 CPUS per SMP node (667 MHz, 1.2 Gflops/cpu), 16 Nodes
- IBM SP at NPACI (SDSC) – Tests yet to be performed!
 - Distributed Shared Memory (DSM)
 - 8 CPUS per node (222 Mhz, 888 Mflops/CPU), 144 Nodes
- IBM SP at ORNL – Tests yet to be performed!
 - Distributed Shared Memory (DSM)
 - 4 CPUS per SMP node (375 MHz, 1.5 Gflops/cpu), 176 Nodes

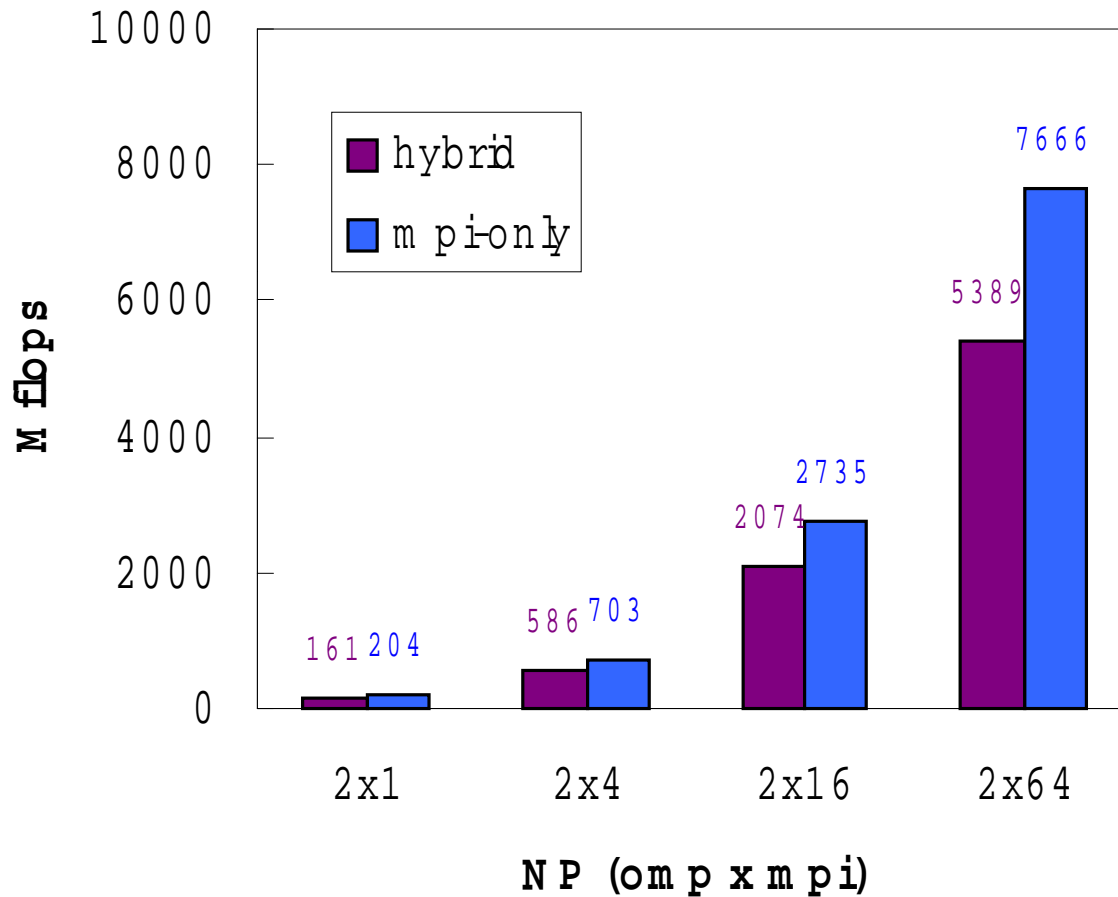
Performance Results

- Most performance results are for the BiCGSTAB matrix solver
 - Solver takes 60-70% of total time for most problems
 - Solver performance is somewhat indicative of the overall performance
- All performance results are based on a 6-component 3D bioremediation groundwater transport problem
 - Problem taken from paper by *Semprini and McCarty, 1992*
 - Base problem size involved about 18,000 elements (110,000 unknowns) – Largest problem (2.5 M nodes, 15M unknowns)
- Results are only presented for SGI Origin 2000 (O2K) and Compaq

MPI vs OpenMP Comparison for BiCGSTAB Solver (O2K)



Hybrid vs MPI-Only Comparision for BiCGSTAB SOLver (O2K)



Compaq Single Node Performance Results

MPI vs OpenMP

NP	MPI Mflops (Speedup)	OMP Mflops (Speedup)
1	216 (1)	103 (1)
2	388 (1.8)	175 (1.7)
4	756 (3.5)	310 (3.0)

MPI vs Hybrid On Compaq

NP (omp x mpi)	MPI Only Mflops	Hybrid Mflops
2 (2x1)	388	175
4 (2x2)	756	397
8 (2x4)	1366	710
8 (4x2)	1366	659
16 (2x8)	2237	1370
16 (4x4)	2237	1118

Conclusions

- Our hybrid implementation did not pay off on both architectures tested
- In order for hybrid implementation to work a full fledged OpenMP implementation may be necessary
 - Loop level parallelism is probably not sufficient
- On both architectures even the single thread OpenMP timings were slower than the single process MPI timings
 - loop level OpenMP parallelism somehow destroys cache benefits
 - An idle thread is created in the beginning which is never used
 - Compiler generates separate subroutines for each parallel do loop
- It appears that better OpenMP compilers may be needed
- Some OpenMP payoff can be expected on the SDSC IBM SP's 8-way SMP nodes
 - Limitation of 4 MPI processes per node (this limitation will ultimately go away)