

SPMD OpenMP vs MPI for Ocean Models
Alan J. Wallcraft
Naval Research Laboratory, Stennis Space Center, MS 39529, USA

Two ocean models, NLOM and NCOM (NRL Layered and Coastal Ocean Models), use domain decomposition to obtain scalability that has been demonstrated on up to 1,152 nodes (Wallcraft and Moore, 1997; Parallel Comp.). In both models, communication between nodes is for the most part performed by calling an application specific programming interface (ASPI), i.e. a set of Fortran subroutines that perform pre-defined high-level communication operations. This allows different low-level “message passing” APIs to be added with relatively little effort. The first two supported APIs were MPI and the SHMEM put/get library, with selection between them being made at compile time via cpp macros.

Domain decomposition is a special case of Single Program Multiple Data (SPMD) parallelization where each node “owns” a distinct sub-domain. When implemented using message passing, each node only holds in memory its own sub-domain plus a large enough “halo” (or set of “ghost cells”) containing copies of values from nearby nodes to allow almost all computation to proceed locally. This splits the problem into a communication phase (updating the halo) followed by a computation phase (with no communication involved). Real applications must handle the additional complexity of I/O and global operations caused by distributing the region across the nodes, but halo exchange is the heart of domain decomposition.

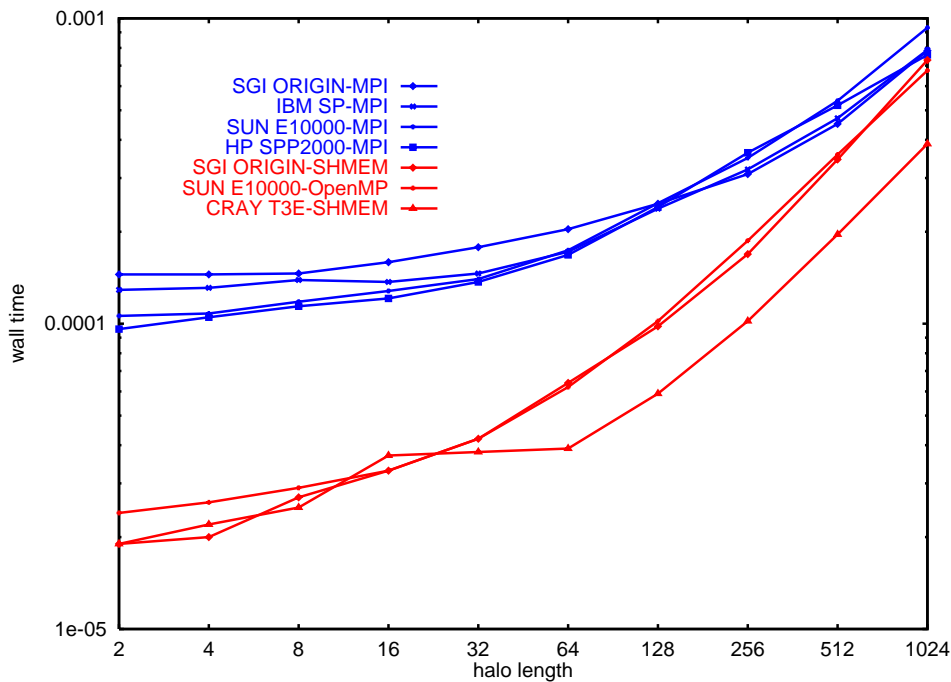


Figure 1: Best HALO times on 16 processors

The HALO kernel benchmark simulates NLOM 2-D halo exchanges on a square N by N sub-domain. Halo exchanges are important operations whenever domain decomposition is used, but HALO can also be treated as a generic low level communication benchmark. It is significantly more realistic than the often used ping-pong benchmark. Figure 1 shows performance for the best HALO implementation of several programming techniques on a range of 16-processor machines. MPI (message passing) performance is similar on all the scalable systems shown and in particular the “shared nothing” IBM SP does about as well as the shared memory machines when using MPI. However, any one of several 1-sided memory methods are always fastest (i.e. have the lowest latency) on machines with global memory. The performance advantage of SHMEM or OpenMP over MPI on this benchmark are significant and translate directly into better scalability when used in real applications such as ocean models. This is why it is advantageous to implement both MPI and SHMEM (say) as alternatives in the same ocean model. MPI runs on all machines, but 1-sided memory methods are significantly faster when they are available.

The SHMEM library was designed for the Cray T3D and T3E, both of which have very fast hardware global barriers. These are so fast that many SHMEM programs issue far more barriers than necessary. On machines without barrier hardware, it is frequently better to synchronize across the minimum necessary number of nodes rather than across all nodes. Figure 2 illustrates this by showing the difference in performance between HALO using a global barrier and using local synchronization. The latter is faster except on the T3E. Even on the T3E, local synchronization is faster beyond about 256 processors.

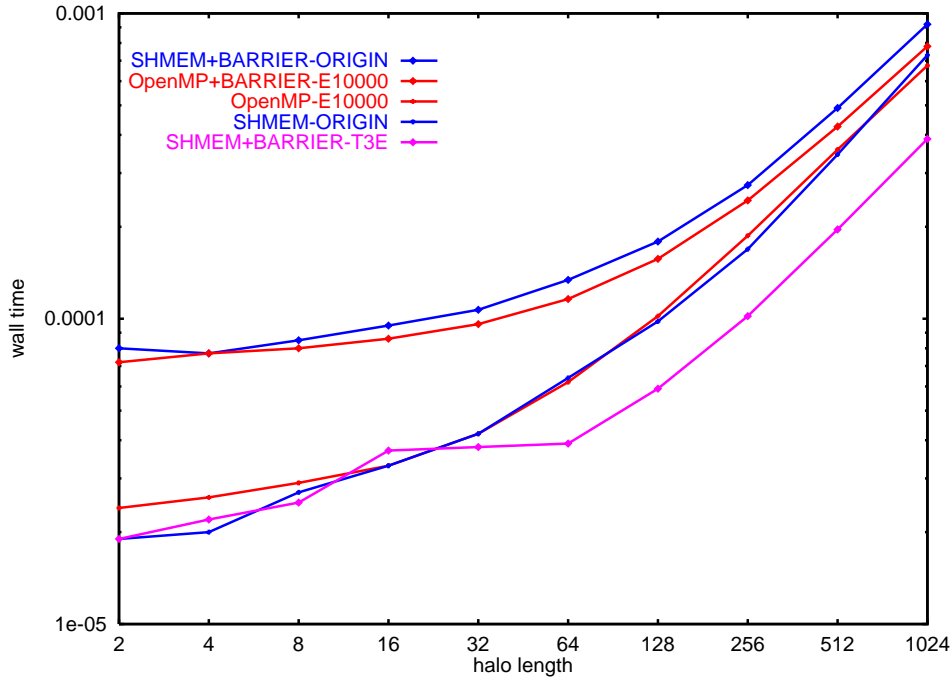


Figure 2: Shared memory HALO times on 16 processors

There is not typically any significant difference in performance at the communication-kernel level between the APIs that implement direct remote memory access (SHMEM, OpenMP, Co-Array Fortran). They all provide a very thin interface on top of the basic memory hardware. However, no single API is available on all machine types with a hardware global memory. In particular, only SGI/Cray supports the SHMEM library. There is much wider support for OpenMP Fortran.

OpenMP can be used in Single Program Multiple Data (SPMD) mode by spawning N threads in the main program and having each thread act from then on similarly to a process in MPI. This is entirely based on OpenMP orphan directives, there are no loop-level directives being used. SPMD OpenMP does not need to use halos for domain decomposition, but in this case OpenMP is being added as an alternative to MPI and so the ocean models are already configured with an explicit array decomposition and halos. Even when implementing domain decomposition exclusively with SPMD OpenMP, it might be advantageous to include halos because they reduce the likelihood of poor performance due to false sharing.

The initial port of NLOM to OpenMP revealed several incompatibilities between thread-based and process-based SPMD coding styles. How these styles differ is summarized in table 1, which includes Co-Array Fortran for future reference.

On a shared memory machine MPI might use shared variables internally, but these are not visible to the application programmer. Under MPI all visible variables are private to each node and communication is via message passing. The simplest mapping from MPI to SPMD OpenMP is to use a “mostly private” scheme, where communication is via shared buffers and everything else is threadprivate. This fits well with the NLOM/NCOM approach of encapsulating communications in an ASPI. Also minimizing and tightly controlling the use of shared variables improves performance on NUMA machines. However, in SPMD OpenMP Fortran the programmer has very little control over which variables are private. In particular, all saved and module variables are always shared and must therefore be replaced by threadprivate common. It is all too easy to inadvertently create a saved variable, for example a local variable ini-

Code Feature	MPI	OpenMP Fortran	Co-Array Fortran
target of compile	single process	multiple threads	multiple images
local variables	private	private	private
saved variables	private	shared	private (or co-array)
module variables	private	shared	private (or co-array)
common variables	private	shared (or private)	private (or co-array)
communication	messages	shared variables	co-arrays
synchronization	implicit	global	global or team (local short cut)
I/O namespace	independent	single and shared	single but private
I/O operations	writes unsafe	unsafe to same unit	safe
Incompatible with Fortran standard	non-blocking calls copy-in/out, I/O	STOP, copy-in/out	none

Table 1: Features of MPI, SPMD OpenMP Fortran and Co-Array Fortran

tialized by a DATA statement is automatically saved and therefore shared. This illustrates the lack of robustness of OpenMP. Many programmers would be surprised to learn that adding a DATA statement to an orphan subroutine is almost guaranteed to break an OpenMP program. This implies that all SPMD OpenMP programs, and many other OpenMP programs, can only be maintained by programmers knowledgeable in OpenMP extensions to Fortran. In contrast, the computational kernels of MPI programs and Co-Array Fortran programs can be maintained and extended by Fortran programmers with no experience with MPI or co-arrays.

Putting saved variables in threadprivate common and adding communication via threadshared common buffers to NLOM was time consuming but routine. Adding support for threaded I/O was much more painful, requiring non-trivial modification to 100's of lines of code. This is because process-based SPMD APIs, such as MPI, have a separate independent I/O namespace per node. Almost all Fortran I/O from a MPI program violates the Fortran standard, but in practice it usually works and (for example) it allows the same record of a READONLY file to be input independently onto each node. OpenMP I/O has a single namespace, so a file can be opened only once (by any one thread) and multiple reads of the same file from different threads will then provide a different record to each thread. In addition, OpenMP I/O to a single file is not thread safe and so all such reads must be in a critical region. OpenMP has obviously been designed as a low level API, giving the programmer control rather than passing responsibility to the compiler. However thread safe I/O is very easy to provide at the compiler level, and very time consuming and error prone for the programmer to implement. The special case of all threads needing to read the same record of a file (into private variables) is so common, and so easy to do in MPI, that it deserves OpenMP compiler assistance. OpenMP has a COPYIN option for some directives and a similar COPYOUT directive qualifier, like that in Cray's CRAFT Fortran, would handle this situation cleanly. For example:

```
!$OMP MASTER
  READ(11) A,B,C
!$OMP END MASTER, WAIT, COPYOUT(A,B,C)
```

Here "A, B, C" are threadprivate variables that are read on the master thread and then copied to all other threads by the COPYOUT clause at the end of the master section. COPYOUT is not available in the current API, but it would greatly simplify support of SPMD I/O in OpenMP. Several other relatively minor additions to the OpenMP API would also aid SMPD programming.

1. COPYOUT has already been mentioned, but an alternative might be to allow the same file to be open simultaneously on multiple units with ACTION=READ. Then each thread can read the same record by opening the file on a different unit per thread. However this represents an extension to the Fortran 90/95 standard, which does not allow the same file to be simultaneously open on two units (although it is perfectly safe for READONLY files).
2. A minimum requirement for SPMD OpenMP to be viable is that STOP on any thread be defined to be a standard STOP for the program as a whole. This seems to be required for conformance with the Fortran standard. In

practice a STOP will typically cause program termination (as expected), but I/O buffers are not necessarily flushed correctly. Since Fortran often makes extensive use of I/O buffers, the lack of a clean STOP capability can render OpenMP unusable in SPMD mode.

3. Treat THREADPRIVATE like SAVE (i.e. applicable to both variable names and the names of common areas) and provide a similar THREADSHARED directive. Then the default status of variables would be much less of an issue because it could be overridden with a compiler directive.
4. Include fully thread-safe I/O, at least as a mandatory compile time option. This is much easier for the compiler writer to provide, either as a thread-safe I/O library or by automatically inserting a critical region around every I/O statement, than the application programmer.
5. Add a richer set of synchronization options. Figure 2 illustrates that a global BARRIER is not sufficient. It is possible for the programmer to write custom synchronization routines in OpenMP Fortran, but they must then always be preceded by a FLUSH directive. If the synchronization routines were part of the API the FLUSH would be unnecessary (implied by the call to a known intrinsic) and the compiler might be able to provide optimized versions. The synchronization primitives from Co-Array Fortran are available as an OpenMP module, and would make an excellent addition to the OpenMP API.

Co-Array Fortran has already been mentioned several times as an alternative Fortran compiler-based SPMD API (Numrich and Reid, 1998; ACM Fortran Forum). It is currently available only on the Cray T3E. As indicated in table 1, Co-Array Fortran's programming model is more similar to MPI (or to SHMEM) than to SPMD OpenMP. For example, all variables are private by default. Co-Array Fortran has been designed so that an "image" can map either onto a process (as in MPI or SHMEM) or onto a thread (as in SPMD OpenMP) or indeed onto a mixture of threads and processes (perhaps on a cluster of SMP nodes). Unlike OpenMP, Co-Array Fortran does not require a global memory system or cache coherence and so it can in principle compete directly with MPI as a "run anywhere" solution. However, this assumes the availability of a Co-Array compiler for shared nothing systems which does not yet exist.

Co-Array Fortran is a very small extension to Fortran 90/95. It is explicitly for SMPD programming, so the single program and all data objects are replicated a fixed number of times. Each replication of the program is called an **image**. One new object, the **co-array**, is added to the language. For example,

```
REAL, DIMENSION(N) [*] :: X, Y
X(:) = Y(:)[Q]
```

declares that each image has two real arrays of size N. If Q has the same value on each image, the effect of the assignment statement is that each image copies the array Y from image Q into its local array X. Co-Arrays cannot be constants or pointers or automatic data objects, and they always have the SAVE attribute unless they are allocatable or dummy arguments. The co-array is the only kind of shared object, and there is no possibility of a pointer from one image to a private object on another image (i.e. there are no global pointers). In OpenMP Fortran there is a relatively weak distinction between shared and private objects, either could be passed to a subroutine using the same dummy argument for example. In Co-Array Fortran the compiler always knows which objects in the local scope are co-arrays.

Array indices in parentheses follow the normal Fortran rules within one memory image. Array indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are always spread over all the images. The programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Global variables are co-arrays spread across all images in Co-Array Fortran, but are shared variables in global memory (not assigned to any particular thread) in OpenMP Fortran. However, this difference is more one of style than substance. It is easy to see that co-arrays could be emulated by shared arrays of higher rank, but conversely a shared array is equivalent to only using the part of each co-array on (say) image 1. For example:

```

COMMON/OPENMP/ X(N,NUM_IMAGES),Y(N,NUM_IMAGES),Z(N,N)
X(:,THIS_IMAGE) = Y(:,Q)
Z22 = Z(2,2)

REAL, DIMENSION(N)[*] :: X,Y
REAL, DIMENSION(N,N)[*] :: Z
X(:) = Y(:)[Q]
Z22 = Z(2,2)[1]

```

In the case of large shared arrays, it would be possible to avoid wasted space by defining a co-array of a derived type with a pointer component and then only allocating an array to the pointer on image 1. This sounds complicated, but is in fact the standard way for Fortran 90/95 to handle an array of arrays (or in this case a co-array of arrays).

Co-Array Fortran I/O is based on a single namespace, but unlike OpenMP Fortran the OPEN must be issued on the image that does the I/O and if multiple images will do I/O to the same unit they must all issue a collective OPEN for that unit. This scheme allows implementation either via independent namespaces (like MPI) or a single namespace (like OpenMP). It is essentially taking the best features of both approaches. In the existing Co-Array I/O scheme only file level operations (OPEN, CLOSE, REWIND, etcetera) are collective, but allowing READ to be collective would handle the “everyone reads the same record” case:

```

READ(11,TEAM=ALL) A,B,C

```

All images in the team perform the identical read and there is implied synchronization before and after the read. This is trivial to implement using independent namespaces (just read the file on all images). If images are implemented as threads, the I/O library could establish a separate file pointer for each thread and have each thread read the file independently or the read could be performed on one thread and the result copied to all others.

Co-Array Fortran provides a richer set of synchronization operations than SPMD OpenMP. The intrinsic subroutine SYNC_ALL() is a global barrier essentially equivalent to OpenMP's BARRIER directive. It is often sufficient, and faster (see figure 2), to only wait for the relevant images to arrive. SYNC_ALL(WAIT) provides this functionality. There is also SYNC_TEAM(TEAM) and SYNC_TEAM(TEAM,WAIT) for cases where only a subset, TEAM, of all images are involved in the synchronization. It is possible to write your own synchronization routines, using the basic intrinsic SYNC_MEMORY. This routine forces the local image to both complete any outstanding co-array writes into “global” memory and refresh from global memory any local copies of co-array data it might be holding (in registers for example). It is therefore equivalent to OpenMP's FLUSH directive. A call to SYNC_MEMORY is rarely required in Co-Array Fortran, because there is an implicit call to this routine before and after virtually all procedure calls including Co-Array's built in image synchronization intrinsics. This allows the programmer to assume that image synchronization implies co-array synchronization. Note that in OpenMP Fortran a FLUSH only applies to shared objects in the local scope, so for example a BARRIER does not imply that all shared objects are synchronized. This is one example of the higher level abstraction provided by the Co-Array Fortran API.

There is a simple mapping from SHMEM put/get library calls onto co-array assignment statements, so adding Co-Array Fortran support to NLOM and NCOM was straight forward and did not require significant changes outside the communication ASPI routines. The full Co-Array Fortran language includes support for “legacy” SHMEM programming practices that would not be recommended to any Fortran 90 programmer, and which limit the implementation options available to a compiler. Removing legacy support leads to a well-defined Subset Co-Array Fortran that has a very natural mapping onto SPMD OpenMP Fortran 90, with Co-Arrays becoming shared OpenMP arrays of higher rank. Therefore NLOM and NCOM now support SPMD OpenMP by translating Subset Co-Array Fortran to OpenMP Fortran using a nawk script as part of the compilation process. The performance of the “native” OpenMP and translated Co-Array Fortran versions of NLOM are virtually identical, so the former has been replaced by the latter (which is much easier to maintain). The effort involved in writing the nawk script was less than that originally expended in manually converting NLOM to support SPMD OpenMP, and the nawk script allows other programs (including NCOM) to be automatically converted. This is a simple pattern matching script which expects idiomatic Co-Array Fortran, and it is only known to work on NLOM and NCOM. What is really required is a full Subset Co-Array Fortran source to SPMD OpenMP source compiler, but the nawk script at a minimum demonstrates that this approach is viable.

The SGI Origin 2000 is a ccNUMA machine that by default places each virtual page in the memory of the processor that first-touches it. This is ideal for NLOM's “mostly private” memory strategy. Figure 3 shows NLOM performance for a fully realistic “NA824” (North Atlantic at 3.5 km horizontal resolution) case. The speed is expressed in Mflops per node, with the floating point operation count always taken to be that from a single processor run (without

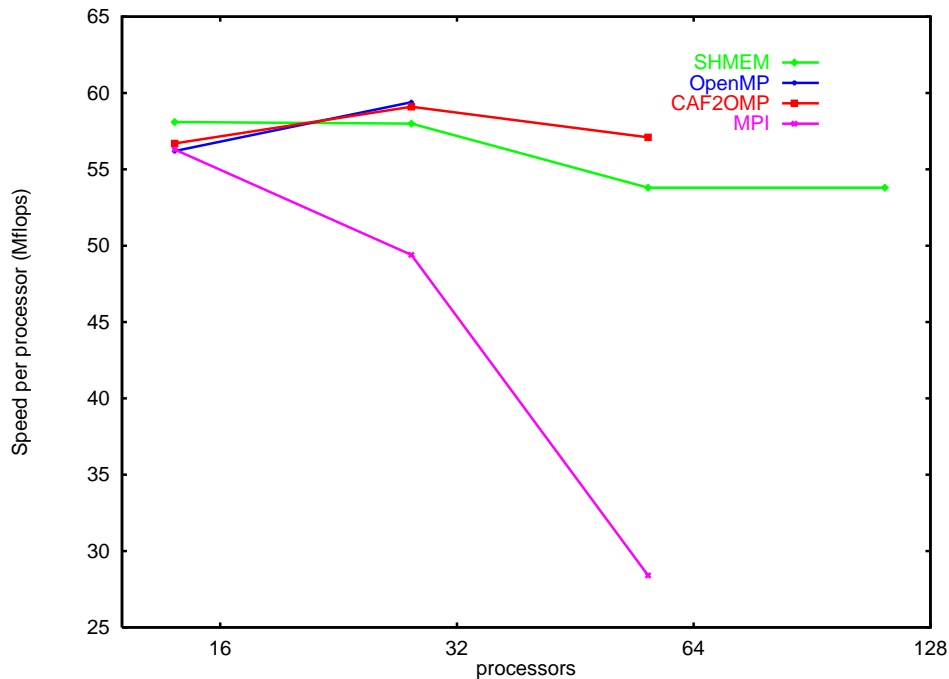


Figure 3: NLOM NA824 Performance on an 195MHz Origin 2000

MADD operations). A constant Mflops per node with increasing node count indicates exact linear scalability, and both SHMEM and OpenMP are close to this ideal. The “CAF2OMP” curve is for Co-Array Fortran translated into OpenMP Fortran, and this almost exactly overlays the native (manually translated) OpenMP curve. On the Origin, OpenMP is marginally faster than SHMEM but the differences are small and probably due to system specific implementation issues (e.g. scheduling of processes vs threads). Both are showing close to linear speedup, in the case of SHMEM from 14 to 112 nodes. No times are reported for 112 nodes for OpenMP because it is now difficult to obtain this many nodes for a single run (the SHMEM 112 node time was obtained when the Origin was first installed, before OpenMP was available). MPI does not scale well above 28 nodes, and this is almost entirely due to a pipelined tridiagonal solver running 10x slower using SGI MPI than using SHMEM on 56 nodes. The same large slowdown does not occur on other machines (e.g. HP/Convex SPP200 or IBM SP) using MPI, and there may well be a set of MPI calls that would allow the pipelined tridiagonal solver to perform better on the Origin. It is characteristic of the large MPI library that different MPI routines perform optimally from one implementation (and one machine type) to the next. One of the advantages of SHMEM, OpenMP and Co-Array Fortran over MPI is that they do not impose a high level abstraction between the machine and the programmer, so they are much less likely to exhibit huge relative performance differences for the same set of operations from machine to machine. However, OpenMP does require attention be paid to how memory maps to cache in order to maximize cache reuse and to minimize false sharing of the same cache line across multiple caches.

SPMD OpenMP Fortran would be a more viable alternative to MPI if it were applicable to a wider range of machine types. The advantage of Co-Array Fortran is that it is potentially applicable to all machine types, and in the mean time it is much easier to support both Co-Array Fortran and MPI (as independent alternatives in the same program) than SPMD OpenMP Fortran and MPI. Co-Array Fortran programs are also intrinsically easier to maintain than SPMD OpenMP (because variables are private by default, and because Co-Array Fortran is a higher level API that places fewer burdens on the programmer). Adding Co-Array Fortran support to OpenMP compilers, either directly or as a separate Subset Co-Array source to OpenMP source step, is a cost effective extension that expands OpenMP's reach to include the many experienced MPI and SHMEM programmers.

This is a contribution to the 6.2 Global Ocean Prediction System Modeling Task. Sponsored by the Office of Naval Research under Program Element 62435N. Also to the Common HPC Software Support Initiative project Scalable Ocean Models with Domain Decomposition and Parallel Model Components. Sponsored by the DoD High Performance Computing Modernization Office.