Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks

Marc Tchiboukdjian*‡, Patrick Carribault†* and Marc Pérache†*
marc.tchiboukdjian@exascale-computing.eu, patrick.carribault@cea.fr, marc.perache@cea.fr
*Exascale Computing Research, Versailles, France
†CEA, DAM, DIF, F-91297, Arpajon, France
‡Université de Versailles-Saint-Quentin-en-Yvelines, France

***Abstract*—With the advent of the multicore era, the number of cores per computational node is increasing faster than the amount of memory. This diminishing memory to core ratio sometimes even prevents pure MPI applications to exploit all cores available on each node. A possible solution is to add a shared memory programming model like OpenMP inside the application to share variables between OpenMP threads that would otherwise be duplicated for each MPI task. Going to hybrid can thus improve the overall memory consumption, but may be a tedious task on large applications.**

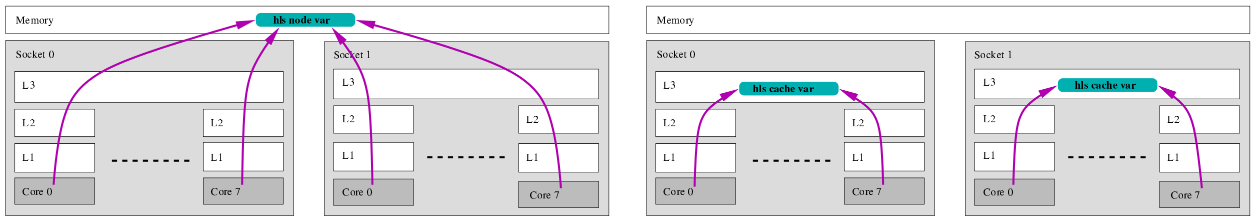
To allow this data sharing without the overhead of mixing multiple programming models, we propose an MPI extension called Hierarchical Local Storage (HLS) that allows application developers to share common variables between MPI tasks on the same node. HLS is designed as a set of directives that preserve the original parallel semantics of the code and are compatible with C, C++ and Fortran languages and the OpenMP programming model. This new mechanism is implemented inside a state-of-the-art MPI 1.3 compliant runtime called MPC. Experiments show that the HLS mechanism can effectively reduce memory consumption of HPC applications. Moreover, by reducing data duplication in the shared cache of modern multicores, the HLS mechanism can also improve performances of memory intensive applications.

***Keywords*-High Performance Computing, Parallel Programming Model, Memory Consumption**

I. Introduction

With the advent of the multicore era, the number of cores per processor and per computational node is increasing. While the total amount of memory per node becomes larger, the memory to core ratio is decreasing. This trend has an impact on the parallel programming models used to exploit parallelism inside scientific applications for High Performance Computing (HPC). Many large-scale scientific applications use MPI as parallel programming model to exploit the performance of a whole cluster, but one drawback of this model is the duplication of data that could be semantically shared among multiple MPI tasks. Indeed, this diminishing memory to core ratio sometimes even prevents pure MPI applications to exploit all cores of each node. One solution to tackle this issue is to add a thread-based programming model like OpenMP inside the application to benefit from shared memory and therefore reduce the overall memory consumption [1], [2]. But going to hybrid may be a tedious task on large applications. The programmer needs to write and to manage two levels of parallelism: one for MPI and one for OpenMP. Furthermore, the Amdahl effect may be large if one wants to dramatically reduce the memory footprint. To minimize data duplication, only one MPI task per node should be created with one OpenMP thread per core on the node. Portions of the code that are not in OpenMP parallel regions are only executed by one core which reduces the potential speedup. This is especially true for MPI communications which are often outside OpenMP parallel regions (called Master-only [1]). The adoption of the Master-only method can be explained by its ease of use and the lack of efficient support of the `MPI_THREAD_MULTIPLE` model in MPI runtime libraries. However this method limits the attainable speedup and may prevent the code to fully utilize the network bandwidth of the machine. To allow this data sharing without the overhead of mixing multiple programming models, we propose an MPI extension called Hierarchical Local Storage (HLS).

The HLS extension aims at reducing the memory footprint by designing a mechanism to share common variables of large-scale scientific applications. To manage the potential performance loss due to a diminished locality when sharing a common variable, one can limit the scope of the MPI tasks sharing the same copy of the data to an element of the memory hierarchy: *e.g.*, NUMA or last level of cache (figure I). When used inside a pure MPI application, the HLS mechanism efficiently reduces the memory consumption without changing the original code performance whereas going to hybrid with the common master-only method might impair performances. Moreover, when used inside a hybrid MPI/OpenMP application, it allows the programmer to decouple data sharing from the programming-model decomposition, *i.e.* the number of MPI tasks per node and the number of OpenMP threads per MPI task. The HLS extension allows the programmer to have an HLS variable with scope **`node`** while its hybrid code has one MPI task per socket or an HLS variable with scope NUMA while its hybrid code has only one MPI task per node.



(a) HLS variable with node scope

(b) HLS variable with cache L3 scope

Figure 1. Two different scopes for HLS variables. The node scope suppresses data duplication on the node but may degrade cache performance if the variable is often modified. The cache L3 scope only reduces data duplication but keep original cache performance.

HLS is a set of directives to mark global variables of the code that can be shared *i.e.*, that may use the same memory location for several MPI tasks. These directives keep the original parallel semantics of the code and ignoring them still produces a correct code. They are compatible with the C, C++ and Fortran languages and with the OpenMP programming model. Because sharing new memory cells adds concurrency, we propose some extra directives to synchronize accesses to these shared variables. We implemented this new mechanism inside a state-of-the-art MPI 1.3 compliant runtime called MPC [3], [4]: the directives are processed by a modified GCC compiler embedded inside the MPC distribution (handling C, C++ and Fortran languages). This compiler generates calls to the runtime library to handle the data sharing. Experiments show that the HLS mechanism can effectively reduce memory consumption of HPC applications. Moreover, by reducing data duplication in the shared cache of modern multicores, the HLS mechanism can also improve performances of memory intensive applications.

The paper is organized as follow. We start by introducing the concept of HLS variables and the associated directives in section II. Then we provide a formal definition of which variables can be made HLS based on the happen-before relationship in section III. Section IV details the implementation. Experimental results based on micro benchmarks and three real applications are given in section V. Finally, we present related works in section VI and conclude in section VII.

II. Informal Definition and Directives

A. Definition

HLS is a mechanism that allows for merging variables sharing the same memory behavior, *i.e.* holding the same value according to the original parallel semantics of the application. Basically, parameters from the input set mainly read and barely written are good candidates for HLS. One example are physics constants used inside numerical-simulation applications. HLS is proposed as an extension to the C, C++ or Fortran source code adding information about the data visibility. The application developer gives the information to the compiler/runtime framework about which variables can be shared by every MPI task. These variables should have the following features:

- **Global variables**
  These data have to be global, whatever the way the language exposes global variables. For example, in C or C++, such variables will have to be outside any functions or to have the `static` qualifier. In Fortran, global variables include common variables, variables with the `save` qualifier and module variables.
- **Same behavior across tasks**
  It is possible to keep the parallel semantics if the variables to use HLS can be shared by every instruction flow. It means that all MPI tasks of the program should access the same value if these accesses were emitted at the same timestamp. A formal definition will be given in section III. Furthermore, to avoid adding too much synchronizations which could reduce performances, these variables should be mainly read and hardly written. However this is not a requirement for the correctness of the resulting code.

If such variables respect the previous constraints, they can be shared across MPI tasks on the same node which reduces memory consumption without adding too much performance overhead (this overhead will be quantified in section V).

Data placement is crucial to reach high performance on a scientific application. Sometimes having one copy of a variable per instruction flow (thread, process, MPI task, OpenMP thread, ...) is one way to improve the locality of memory accesses to this variable. For example, a variable whose value is often modified benefits from being private to each instruction flow since coherency traffic is removed and the variable can stay in the caches. Another example is due to non uniform memory access (NUMA) effects. If a variable is private to each instruction flow, it can benefit from the first touch policy and be allocated on the closest NUMA node. Latency and bandwidth when accessing this variable is improved since it always comes from the closest NUMA node.

The HLS mechanism proposed in this article reduces data duplication but may degrade performances as a side effect since variables are no longer private to an instruction flow. To avoid reducing data locality while still keeping memory consumption under control, an HLS variable can be declared with a scope corresponding to the memory hierarchy of the

```
int a,b ;
#pragma hls node(a)
#pragma hls numa(b)

void f() {
  ...
#pragma hls single(a)
{
  // executed by one instruction flow per node
  a = 4 ;
}
  ...
#pragma hls single(b)
{
  // executed by one instruction flow per NUMA node
  b = 2 ;
}
  ...
}
```

Listing 1. Modifying HLS variables with the pragma **single**

platform (figure I): *e.g.* node, NUMA, last level of cache, etc. An HLS variable will be shared only by the instruction flows sharing the specified level of the memory hierarchy. The application developer can thus obtain the right tradeoff between memory consumption and performance.

B. Directives

HLS is designed as directive extensions for C, C++ and Fortran programming languages (**#pragma** in C and C++ or lines starting with `!$` in Fortran). This solution keeps the semantics of the original parallel code. Thus if the directives are not parsed and recognized by the compiler, the application behaves as if there were no extensions at all.

The HLS mechanism is divided into two directive categories. The first one marks a variable as HLS and specifies the scope of this storage. The second category handles the new concurrent accesses.

1) Changing Data Visibility: These new directives should be composed of 2 elements: (i) the data scope and (ii) the variables whose scopes have to be updated. An optional third argument is related to the level of the corresponding scope (*e.g.*, the cache level if the target processor has multiple level caches). To register variables as HLS, we adopt a syntax close to the OpenMP attribute `threadprivate` [5]. The syntax is as follows:

```
#pragma hls scope(var1, var2, ..., varN) [level(L)]
```

The **scope** attribute represents the target data scope and may have the following values:

- **node**
  The variables will be shared by every MPI task running on the same node.
- **numa**
  There will be one copy of every variable per NUMA node. On current Xeon architecture (Nehalem/Westmere), one NUMA node is actually a socket. This scope accepts the **level** clause including the level of the NUMA node where this data should be duplicated.

```
int a,b ;
#pragma hls node(a)
#pragma hls numa(b)

void f() {
  ...
#pragma hls barrier(a,b)
  ... // no access to a and b
#pragma hls single(a) nowait
{
  // executed by one instruction flow per node
  // no access to b
  a = 4 ;
}
  ... // no access to a and b
#pragma hls single(b) nowait
{
  // executed by one instruction flow per NUMA node
  // no access to a
  b = 2 ;
}
  ... // no access to a and b
#pragma hls barrier(a,b)
  ...
}
```

Listing 2. Modifying HLS variables with the pragma **single nowait**

- **cache**
  One copy per cache. This scope accepts the **level** clause (from 1 to the last level of cache (**llc**)).
- **core**
  One copy per physical core. Hyperthreaded processors benefit from this level because there will be one copy of each variable per physical core, allowing sharing among hyperthreads scheduled on the same core.

The second clause needed for this extension is a list of variables. Every element of this list should be an already defined variable but it should not have already been accessed (read or write). These constraints are exactly the one of the `threadprivate` directive.

2) Handling Concurrent Accesses: Adapting the data visibility has a large impact on the data concurrency. Indeed, with the scope **node**, a variable will be shared by every MPI task located on the same computational node. It means that the write accesses have to be handled to avoid data races. For this purpose, we propose the following new directive (inspired from the **single** workshare construct in OpenMP [5]):

```
#pragma hls single(var1, var2, ..., varN) [nowait]
```

This directive accepts a block of instructions: these instructions will be executed only by one MPI task depending on the data visibility (HLS scope) of the variables located inside the argument part. Notice that these variables have to be marked as HLS and need to have the same HLS scope. Otherwise, the compiler will generate an error. This directive implies an implicit barrier at the beginning and at the end, blocking every MPI task belonging to the corresponding scope. Inspired from the OpenMP standard, the **nowait** keyword allows the MPI tasks to skip the region encapsulated by the pragma **single** instead of waiting for the entered MPI task to finish the execution of the block. This keyword removes the implicit two barriers at the beginning and at the end of the region.

To complete this directive set, we propose an explicit barrier:

```
#pragma hls barrier(var1, var2, ..., varN)
```

It synchronizes every MPI task belonging to the largest HLS scope of the variables in the list (node is the largest scope and core the smallest).

Examples in listings 1 and 2 illustrate how to use the pragmas **single** and **barrier**. The write operations on the two HLS variables `a` and `b` are protected by a `pragma` **hls single** so that the common copy is written only once. In listing 1, synchronizations between the execution flows are ensured by the **single** whereas in listing 2, synchronizations are ensured by two explicit barriers (**#pragma hls barrier**) before the first and after the last **single**. This reduces the number of synchronizations by a factor of 2. Notice that the two versions are not equivalent. In the second version with explicit barriers, we cannot use the value of variable `a` outside of the region encapsulated by the **#pragma hls single**`(a)` and the same for variable `b` as these variables may not have been updated yet.

C. Advantages/Restrictions

There are two main advantages for this extension.

1) Memory gain
   Moving upwards the visibility of a variable remove data duplication. This leads to a memory gain of a factor of up to the number of cores on a computational node.
2) Cache effects
   Instead of accessing multiple memory locations holding the same value, every MPI task accesses the same memory block and share it inside shared caches (*e.g.*, L3 cache on Nehalem/Westmere architecture). This may lead to a performance improvement if those variables are stressed. For example, one access by a MPI task may benefit another MPI task if the first one retrieves the correct cache lines to the shared cache.

It should be noted that this extension does not violate the original semantics *i.e.*, a compiler unaware of these directives can ignore them and should generate a correct code if the program was correct without them.

The HLS features have the following restrictions.

- Only global variables can have an HLS scope. However an HLS global variable can point to heap-allocated memory with a proper use of the single directive around memory allocation/deallocation (see listing 4 and section IV-C).

```
#define RES (1024)
double table[RES];
#pragma hls node(table)
// there is only one copy of the
// array table per node

int main( int argc, char **argv ){
  double *mesh;
  unsigned int X, Y, Z, T, x, y, z, t;
  int rank, size;

  X = atoi( argv[1] ); Y = atoi( argv[2] );
  Z = atoi( argv[3] ); T = atoi( argv[4] );

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );

  // allocate memory and initialize
  // with random real in [0,1]
  init_mesh( &mesh, X, Y, Z );

#pragma hls single(table)
{
  // load table from file
  // this function is only executed
  // by one MPI task per node
  load_table();
}

  for( t = 0; t < T; ++t ){
    MPI_Barrier( MPI_COMM_WORLD );
    for( x = 0; x < X; ++x )
      for( y = 0; y < Y; ++y )
        for( z = 0; z < Z; ++z )
          compute_cell( &mesh[x][y][z] );
  }

  free( mesh );
  MPI_Finalize();
}
```

Listing 3. Physical constants

- The main HLS directive can be put where an OpenMP `threadprivate` directive could be added (between variable definition and declaration, type has to be complete, etc.) The HLS single directive can be put where an OpenMP **single** directive could be added (where a statement could fit according to the language standard). For more details, please see the OpenMP reference manual [5].
- All or none MPI tasks should execute a single or barrier directive. This is similar to MPI and OpenMP collective operations.

D. Examples

We highlight how to use the HLS pragmas through two examples: one involving physics constants and the other involving matrix multiplications with a common matrix. These examples are inspired from two real applications.

1) Mesh update with a common table: This example illustrates how the HLS mechanism can reduce memory consumption by sharing the same copy of a variable between multiple MPI tasks running on a common node.

```
double *A, *B, *C;
#pragma hls node(B)

int main( int argc, char **argv ){
  int rank, size, t, N, K, M, T;
  N = atoi( argv[1] );  M = atoi( argv[2] );
  K = atoi( argv[3] );  T = atoi( argv[4] );
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );

  // allocate memory and initialize matrices A, B
  // matrices A and C are distinct for each MPI task
  // whereas matrix B is common to all MPI tasks and
  // thus initialized only by one task per node
  init_matrix( &A, N * K );
  init_matrix( &C, N * M );
#pragma hls single(B)
{
  init_matrix( &B, K * M );
}

  MPI_Barrier( MPI_COMM_WORLD );

  for( t = 0; t < T; ++t ){
    cblas_dgemm( CblasRowMajor, CblasNoTrans,
     CblasNoTrans, N, M, K,
     1.0, A, K, B, M, 1.0, C, N );
    MPI_Barrier( MPI_COMM_WORLD );
  }

  free( A ); free( C );
#pragma hls single(B)
{
  free( B );
}
  MPI_Finalize();
}
```

Listing 4. Matrix multiplications with a common matrix

A good candidate to become an HLS variable is an array storing physical constants. These constants are common to all MPI tasks of the application and are not modified throughout the run. These tables can be as big as several hundreds megabytes and having only one copy per node significantly reduces memory consumption.

The code of listing 3 declares a global array `table` with the **`#pragma hls node`** (lines 2-3). This array is first initialized from a file by only one task per node thanks to the **`#pragma hls single`** (lines 23-29). It is then used by all MPI tasks to update a 3D mesh for `T` time steps (lines 31-37).

2) Matrix multiplication with a common matrix: This example illustrates how the HLS mechanism can improve performance by reducing shared cache misses. In this example, at each time step, every MPI task updates its matrix by performing a matrix multiplication with a common matrix. Sharing the same copy of this common matrix saves space in the last level of cache common to all cores of the same processor. Depending on their size, all matrices (the common one plus the private ones) may now all fit in the shared cache and be reused from one time step to the next without any additional cache misses.

The code of listing 4 declares 3 matrices `A`,`B` and `C`. Matrix `B` is common to all MPI tasks and thus declared with the HLS scope **`node`**. Each MPI task repetitively performs the operation $C \leftarrow A \times B + C$. The allocation and deallocation of memory for matrix `B` is encapsulated in a **`#pragma hls single`** so that it is processed only by one MPI task per node.

III. Formal Definition

A. Happens-before Relationship

To give a formal definition of which variables are eligible to use HLS, we first need to introduce the happens-before relationship $\prec$ between two events a and b of a parallel program [6]. We say that $a \prec b$ if a is executed before b in all schedules compatible with the partial order defined by the synchronizations of the parallel program. If neither $a \prec b$ nor $b \prec a$, we say that a happens in parallel with b, $a \parallel b$.

```
// MPI task of rank 0         // MPI task of rank 1
a();                          b();
MPI_Send( ..., 1, ...);       MPI_Recv( ..., 0, ...);
c();                          d();
```

For example, in the MPI program above, the function call `a` in the MPI task of rank 0 happens before the function call `d` in the MPI task of rank 1 because a message is sent after the execution of `a` on rank 0 and received before the execution of `d` on rank 1. Similarly, the function call `c` in the MPI task of rank 0 happens in parallel with the function calls `b` and `d` in the MPI task of rank 1 because the MPI program does not add any synchronization between these two events. In a real execution, `c` may be executed before, at the same time or after `b`. We can deduce two more precedence relations, $a \prec c$ and $b \prec d$, as these function calls are executed by the same MPI task in the sequential order defined by the program.

B. HLS Variables without Additional Synchronizations

We now formally define which variables are eligible to use HLS. We first define when a read to a variable is coherent. If all reads of all MPI tasks on a variable are coherent then this variable is eligible to use HLS.

We consider a read operation r that returns the value $v(r)$. This read operation is coherent if the two following conditions on write operations w to the same variable are satisfied.

1) All write operations to the same variable that happens in parallel with r write the same value as r, *i.e.*

$$\forall w \; w \parallel r \implies v(w) = v(r).$$

2) All write operations w to the same variable that happens before r and for which there is no other write

operation to the same variable in between write the same value as r, *i.e.*

$$\forall w \ (w \prec r \ \wedge \ \nexists w' \ w \prec w' \prec r) \implies v(w) = v(r).$$

It is easy to see that if one of these two conditions is not satisfied, there exists a schedule compatible with the partial order defined by the synchronizations of the MPI program in which the delinquent write happens just before the read operation that will thus return a wrong value.

Variables satisfying the conditions 1 and 2 can be shared between MPI tasks without adding any synchronization in the MPI program. If these conditions are not satisfied, it may still be possible for a variable to use HLS by adding synchronizations in the program.

C. HLS Variables with Additional Synchronizations

A necessary condition for a variable to be eligible to use HLS is that

3) at least one write operation considered in conditions 1 and 2 writes the same value as the read operation, *i.e.*

$$\exists w \ \{w \parallel r \vee (w \prec r \wedge \nexists w' \ w \prec w' \prec r)\} \wedge v(w) = v(r).$$

If this condition is not satisfied, at least one of those writes will happen just before the read and the resulting value will be incorrect. In this case, it may be possible to make the variable HLS if one can add synchronizations such that one of the writes that do not satisfy condition 3 happens before the read operation and no other writes on the same variable happen in between. If these new synchronizations conflict with existing synchronizations, *i.e.* adding them would create a cycle in the precedence graph defined by the happens-before relationship, the variable cannot be made HLS.

We do not provide a generic way to add such synchronizations but the pragma **`single`** allows to make a variable eligible to use HLS in some specific cases that happens often in MPI programs due to their SPMD nature. If each MPI task executes the same sequence of write operations to a variable, *i.e.* the same number of write operations with values in the same order, we can encapsulate each of those write operations with **`single`** pragmas. The **`single`** pragma behaves as a barrier in term of synchronizations so adding it may conflict with existing synchronizations. In the other case, the variable is eligible to use HLS. Indeed, any read operation on this variable will be between two pragmas **`single`** where the value of the variable is coherent. The only preceding write operation, for condition 2, is the one encapsulated in the pragma **`single`** and has the same value than the read operation. For condition 1, there is no write operations that happens in parallel as the following write operation is in the next single and thus a barrier is placed between the read and the write operation.

IV. Implementation

The HLS mechanism requires a compiler/runtime cooperation. The compiler part of the implementation has been done in the GCC compiler and supports the languages C, C++ and Fortran. A new option, `-fhls`, has been added to activate the new HLS directives. The compiler detects and parses the pragmas, modifies the code and the visibility of the variables accordingly, and generates calls to runtime functions.

The runtime part of the implementation has been done in MPC, a state of the art MPI 1.3 and OpenMP 2.5 unified runtime [3], [4]. An interesting feature of MPC is that MPI tasks are executed inside user-level threads instead of processes unlike classical MPI libraries like Open MPI or MPICH2. Thus, in MPC, MPI tasks on the same node share by default the same address space. Therefore the HLS functionalities are easier to implement. However, these functionalities are still compatible with standard process-based MPIs (see section IV-C).

A. Data Visibility

To handle the data visibility of HLS variables, the parsing and the code generation of the compiler need to be adapted. In the parsing step, when an HLS pragma is encountered, the compiler checks that the corresponding variable is global and has not been used yet and then flags the variable like a thread local storage (TLS) variable but with a TLS type corresponding to the desired scope: node, numa, etc.

```
void *hls_get_addr_node(size_t mod, size_t off);
void *hls_get_addr_numa(size_t mod, size_t off);
```

In the code generation step, when an HLS variable is used, a function call according to the scope is inserted to get the address of the variable. A variable is identified by the two arguments: the module which corresponds to the program or the library where the variable is declared and its offset in the memory area. The code is transformed as follows.

```
int a;                    int *ptr_a;
#pragma hls node(a) ⇒    ptr_a = hls_get_addr_node(0,0);
a = 3;                    *ptr_a = 3;
```

The linker is then responsible for filling the right module id and the offset as arguments.

The MPC runtime needs to allocate memory for the HLS variables and to implement the `hls_get_addr_<scope>` functions. Each MPI task maintains a private array of pointers, one for each scope, pointing to an array storing information on the currently loaded modules (figure 2). This array is handled like regular TLS variables [7]. Two MPI tasks on the same scope, *e.g.* two MPI tasks on the same node or two MPI tasks on the same NUMA node, point to the same module array and thus share variables with this scope. When a MPI task wants to migrate, it first needs to check that it has encountered the same number of single and barrier directives as the destination. If it is the case, the

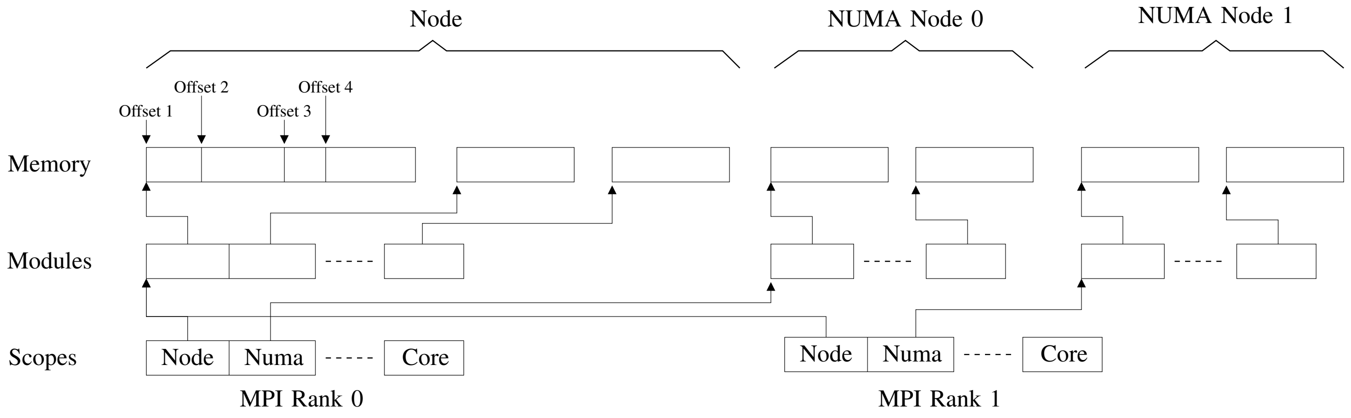


Figure 2. Memory layout of HLS structures. Each MPI task has its own array of scopes. For the scope **node**, two MPI tasks on the same node share the structures and thus share variables with scope **node**. If they are on two different NUMA nodes, each has its own structure for the **numa** scope with different variables.

task can migrate and then use the destination HLS variables by updating its private array of points. HLS variables are linked to the architecture, therefore they do not move. In MPC, each MPI task is pinned to a core by default. A task can only migrate is specified by the application programmer using the call `MPC_Move` [3].

Memory for a module is allocated and initialized at the first call to the get address function `hls_get_addr_<scope>`. It is then straightforward to get the address of a variable

```
void *hls_get_addr_<scope>(size_t mod, size_t off){
  // allocate and initialize memory if first use
  return hls[<scope>][mod] + off;
}
```

To handle concurrency when allocating and initializing memory for a module or adding a new module in the module array, a lock is associated to each module and each module array.

B. Synchronizations

The HLS directives specify tthreee kinds of synchronizations: **barrier**, **single** and **single nowait**.

For the pragma **barrier**, the compiler parses all variables in the list, checks if all of them are HLS variables and in this case generates a function call to the runtime with, as argument, the largest scope of all variables in the list. The **core** scope is the smallest and the **node** scope the largest. For example, if the list contains a variable with scope **node**, the barrier will synchronize all MPI tasks running on the same node. For all scopes except **numa** and **node** we implement a simple flat algorithm with a counter and a lock. For the larger scopes, we implement a shared-cache aware barrier: all MPI tasks in the same **llc** scope synchronize first and only one of them goes to the next scope. This way, locks and counters stay in the shared cache and all synchronizations at the **llc** scope happen in parallel.

The pragma **single** behaves as if there were a barrier before and after the execution of the encapsulated code block. To reduce the overhead of this pragma, it is implemented in a single modified barrier. The barrier algorithm is similar to the pragma **barrier** except the last MPI task entering the barrier executes the code block before releasing the others tasks waiting on the barrier. To do so, we need to modify the code at compile time. We add an if condition around the code block driven by a call to the runtime implementing the first part of the barrier `hls_single()`. Every MPI task waits inside this function except for the last one for which the function returns true. This last task executes the code block and then releases the other MPI tasks thanks to a call to the function `hls_single_done()` added at the end of the code block. Then `hls_single()` returns false and the other MPI tasks do not execute the code block. The following example illustrates how the code is modified.

```
int a;
#pragma hls node(a)
                               if(hls_single(node)){
#pragma hls single(a)   ⇒       f(&a);
{                                hls_single_done(node);
  f(&a);                       }
}
```

The pragma **single nowait** is handled similarly except there is no need for a barrier before or after. The code block is encapsulated in an if condition driven by a call to the runtime that returns true only for one task. In this case, it is the first MPI task entering the **single nowait** that executes it. The following example illustrates how the code is modified.

```
int a;
#pragma hls node(a)
                            if(hls_single_nowait(node)){
#pragma hls                   f(&a);
 single(a) nowait    ⇒      }
{
  f(&a);
}
```

To ensure that the code inside the pragma **single nowait** is executed only once, a counter is associated to each scope. Each MPI task maintains counters equal to the number of such pragmas encountered for each scope. When entering a **single nowait**, a MPI task increments its counter and if this counter is ahead of the counter associated with the scope, the scope counter is incremented too and the MPI task executes the code encapsulated in the single.

C. Implementation Details for Process-Based MPIs

In process-based MPI implementations, MPI tasks are UNIX processes and have different address spaces. To be able to share variables and use shared-memory synchronization algorithms, all HLS variables and the corresponding structures must be allocated in a memory segment shared by all processes of the same node. Additionally this shared memory segment should start with the same virtual address for all processes on the node. This can be achieved using the `mmap` primitive which allows to allocate memory at a specified virtual address. This technique is used to implement the isomalloc of PM^2 [8]. The rest of the implementation is similar to the case of thread-based MPIs except for the handling of heap-allocated memory. When a HLS variable is holding a pointer to heap allocated memory like for example the code of listing 4, one must ensure that this memory has been allocated in the shared memory segment. A possible solution is to overload dynamic memory allocations (for example with the `LD_PRELOAD` mechanism) and allocate memory in the shared memory segment when the call is inside a **single** directive.

V. Experimental Results

The experimental results are split into two parts. In the first part, we show that, although performance improvement is not the main focus of the HLS mechanism, there is still some gain due to a reduced consumption of shared cache space. In the second part, we present the main contribution of the HLS mechanism: memory consumption reduction.

A. Cache Footprint Reduction

Modern multicore processors often have a last level cache that is shared among all the cores. In a pure MPI execution with one MPI task per core, MPI tasks compete against each other for shared cache space. The fact that the cache is shared does not improve performance as MPI tasks do not share data. The HLS mechanism can lower pressure on the shared cache space by reducing the number of copies of the same data from one per core to one per shared cache. With the growing number of cores per processor, this can reduce the shared cache consumption of HLS data by a factor of 8 or more.

To highlight this phenomenon, we used a node with 4 Nehalem-EX processors (Intel Xeon X7550 @2.00Ghz). Each processor have 8 cores sharing a last level cache of 18MB. Note that, on this node, there is one processor per NUMA node thus the **hls numa** scope and the **hls cache level(llc)** scope are identical. In the following, we used the scopes **hls node** and **hls numa**.

	Parallel efficiency					
	without update			with update		
mesh size	small	medium	large	small	medium	large
without HLS	37%	39%	40%	30%	37%	40%
HLS node	94%	93%	99%	65%	87%	95%
HLS numa	94%	93%	99%	88%	92%	97%

Table I
Performance improvement due to cache footprint reduction on the mesh update benchmark on 4 Nehalem-EX processors.

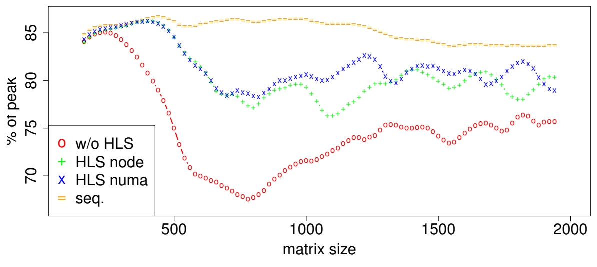
In those experiments, we use the two examples presented in section II-D: mesh update with a common table and matrix multiplication with a common matrix.

1) Mesh update benchmark: In this benchmark (*cf.* section II-D1) each MPI task owns a 3D sub-domain of variable size: $50\times50\times50$ for the small setting, $100\times100\times100$ for the medium setting and $200 \times 200 \times 200$ for the large setting. A mesh cell is represented by a floating point in double precision resulting in a sub-domain size of roughly 1MB, 8MB and 60MB respectively. At each time step, each mesh cell is updated using a value interpolated in a common 2D table of size 1000×1000. The table has a size of roughly 8MB. To mimic an irregular access pattern, this table is accessed uniformly at random.

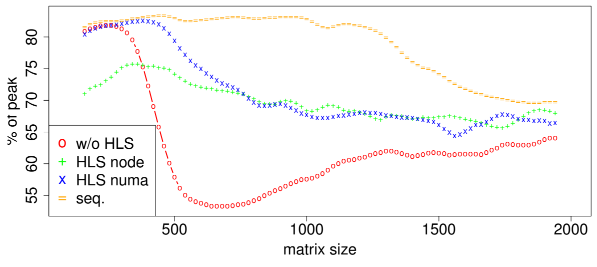
In a regular MPI program, this table will be duplicated 8 times per processor and thus all these copies cannot fit in the 18MB shared cache. In the HLS version, this table will not be duplicated and thus can fit in the shared cache. If the table stays in the last level of cache between time iterations, access times to the table should be reduced except for the first iteration.

To emphasize the difference between the scopes **node** and **numa**, two versions of the benchmark have been developed. In the no-update version, the values in the table are initialized once and do not change over time. In the update version, the table is modified at each time step. This modification is encapsulated in a `pragma` **hls single**. In the no-update version, we do not expect to see a big difference between scopes **node** and **numa** since the table should stay in the shared cache between iterations. In the update version the table is modified between each time step and thus with the **node** scope the table will be invalidated in all shared caches except the one of the core doing the modification. With the **numa** scope, one core per shared cache is modifying the table and thus all copies stay valid between time iterations.

Table I compares the parallel efficiency (the ratio between the speedup and the ideal speedup) of the regular MPI program (without HLS), the MPI program with HLS scope **node** and the MPI program with HLS scope **numa**. This is a weak scaling study, the sequential program only computes one sub-domain. The parallel efficiency is computed as $t_{\text{par}}/t_{\text{seq}}$. We expect the efficiency to be lower than 1 since the sequential program can fully utilize the last level of cache and the memory bandwidth of the processor whereas the parallel program shares these resources between 8 cores.



(a) No-update version



(b) Update version

Figure 3. Performance improvement due to cache footprint reduction on the matrix multiplication benchmark on 4 Nehalem-EX.

Informally, an efficiency close to 1 indicates that there is no contention on the shared resources. The regular MPI program without HLS has a relatively low efficiency between $30-40\%$ since its working set cannot fit in the last level of cache due to the duplication of the common table. The two HLS versions greatly improve the parallel efficiency of the MPI program with efficiencies close to 100%. As expected, the **`numa scope`** behaves better than the **`node scope`** on the update version which confirms the interest of having different scopes. The efficiency of the **`node scope`** on the small setting of the update version is lower since the working set of the sequential program can fit entirely in cache in this case (less than 9MB). The cache misses induced by the invalidation of the table during its update significantly reduce the performances.

2) Matrix multiplication benchmark: In this benchmark, each MPI task repetitively performs a matrix multiplication $C \leftarrow A \times B + C$ where the matrix B is common to all MPI tasks. Similarly to the mesh update benchmark, we expect to see a performance gain by using the HLS mechanism on the common matrix thanks to a reduced pressure on the shared resources. Again, two versions have been implemented. In the update version, the matrix B is modified after each time step. In contrary to the previous benchmark, the access pattern to the HLS variable is regular. We use the MKL implementation of the BLAS to compute the matrix multiplication.

Figure 3 compares the performance of the sequential program, the regular MPI program (without HLS), the MPI program with HLS scope node and the MPI program with HLS scope numa. This is a weak scaling study, the sequential program computes the same matrix multiplication as one MPI task of the parallel program. Since the sequential program does not share the last level of cache and the memory bandwidth, it shows the best performances. For both versions (no-update and update), the performance of the regular MPI program is significantly lower than the sequential program except for very small matrix sizes where all matrices A, B and C fit in cache simultaneously. The two HLS programs show performances closer to the sequential program. For small matrix sizes, the HLS programs and the sequential program have the same performances. Performances start to fall off later for the HLS program than the regular MPI program since the smaller working set (matrix B is not duplicated) of the HLS program can fit in cache longer. The gap between the regular MPI program and the HLS programs is maximal when the regular MPI program starts to go off cache and the gap reduces with growing matrix sizes. Interestingly, even for working sets significantly larger than the last level of cache, HLS programs still have a performance improvement. We checked this phenomenon for matrices up to size 4000. We conjecture that the HLS programs are less impacted by the memory bandwidth since MPI tasks access the same part of matrix B approximately at the same time and that the shared cache can capture this inter MPI tasks locality even for working sets larger than the cache. For the update version, the HLS scope numa program performs better than the HLS scope node for small matrix sizes when the matrix B can stay in cache across time iterations. Indeed, in the HLS scope node program, the update of matrix B invalidates the cached version loaded by the previous time iteration.

Since these two benchmarks are inspired from two real applications, we also measured the performance gains of making HLS these common variables in the original applications. However, the improvements were slim since it only affects a small part of all operations of a time iteration.

B. Memory Footprint Reduction

In this section, we present the main purpose of the HLS mechanism: memory consumption reduction. We use three applications: EulerMHD [9], Gadget-2 [10], which has been selected as a PRACE application [11] and Tachyon [12], which is part of the SPEC MPI2007 benchmark. These applications run on an Infiniband cluster with up to 92 nodes equipped with 2 Intel Core2 quadcore (Intel Xeon E5462 @2.80Ghz) for a total of 8 cores per node. We can thus expect a memory reduction of a factor 8 for HLS scope

# cores	MPI	time (s)	avg. mem. (MB)	max. mem. (MB)
256	MPC HLS	145	651	672
	MPC	146	1570	1590
	Open MPI	135	1715	1786
512	MPC HLS	73	490	550
	MPC	73	1417	1466
	Open MPI	68	1573	1732
736	MPC HLS	51	455	531
	MPC	51	1375	1448
	Open MPI	47	1574	1796

Table II
EXECUTIOIN TIME AND MEMORY CONSUMPTION FOR EULERMHD

# cores	MPI	time (s)	avg. mem. (MB)	max. mem. (MB)
256	MPC HLS	1540	703	747
	MPC	1540	938	988
	Open MPI	1438	1731	1742

Table III
EXECUTION TIME AND MEMORY CONSUMPTION FOR GADGET-2

node variables. We compare the execution time and the memory consumption of the applications run with MPC, MPC with the HLS mechanism enabled and Open MPI [13]. The memory consumption of the application plus the MPI runtime is measured every 0.1s on each node. With these applications, the memory consumption is stable after a start-up phase thus only the average over time is reported. This measure is then averaged on all nodes, the maximum on all nodes is also presented.

1) EulerMHD: EulerMHD is a pure MPI code that solves both the Euler and the ideal magnetohydrodynamics (MHD) equations at high order on a two dimensional Cartesian mesh. In these experiments, we use a mesh of size 4096×4096. The equation of state of the gas is stored in a two dimensional table. This table allows to calculate the pressure of the gas from the density and the internal energy. This table is constant over all MPI tasks and can thus use HLS. We added in the original code one pragma to declare this table HLS with scope node and one pragma **`single`** around its initialization. As this table consumes approximately 128MB of memory, we can expect a memory gain of $7 \times 128 = 896MB$ per node. Table II compares the memory consumption of EulerMHD with MPC, MPC with the HLS mechanism enabled and Open MPI. One can remark that the MPC runtime consumes between 100 and 300MB less memory than Open MPI and this gap grows with the number of cores. This can be explained by a less aggressive policy on communication buffers. Moreover, we observe the expected memory gain (around 900MB) when the HLS mechanism is enabled. The overhead of the HLS mechanism on the execution time (due to address computation functions and additional synchronizations) is negligible.

2) Gadget-2: Gadget-2 is a pure MPI code for cosmological N-body smoothed particle hydrodynamic simulations. When using periodic boundary conditions, the force and the potential need to be corrected due to the infinite number of particles. These corrections are obtained by Ewald summation and computed by trilinear interpolation from a precomputed table. This table is constant over all MPI tasks and can thus use HLS. We added in the original code one pragma to declare this table HLS with scope node and one pragma **`single`** around its initialization. This table has a size of approximately 33MB. On an 8-core node, the memory gain should be around $7 \times 33 \approx 230$MB. Table III compares the memory consumption of Gadget-2 with MPC, MPC with the HLS mechanism enabled and Open MPI. Conclusions are similar to the EulerMHD application. MPC consumes less memory than Open MPI. We well observe the expected memory gain. The overhead of the HLS mechanism on the execution time is still negligible.

# cores	MPI	time (s)	avg. mem. (MB)	max. mem. (MB)
736	MPC HLS	83	748	931
	MPC	88	4786	4975
	Open MPI	89	4885	5118

Table IV
EXECUTION TIME AND MEMORY CONSUMPTION FOR TACHYON

3) Tachyon: Tachyon is a ray tracing application which supports two parallelism models: pure MPI and hybrid MPI plus threads. We only use here the pure MPI version. Work is decomposed by giving an identical number of rays to each MPI task. The largest data structures used are the scene and the resulting image. The scene mainly contains a number of objects with their associated textures. The image is an array of RGB pixels whose size corresponds to the resolution. The scene is replicated across all MPI tasks since it is hard to predict what part of the scene a ray will access when bouncing between objects. The image is also duplicated for code simplicity reasons. The whole image is only used by MPI task 0 when receiving all parts of the image computed by the other tasks. The other MPI tasks only need a small part of the image to store the color of the rays they computed. As the scene is not modified during rendering, it can use HLS. Moreover, the image can also use HLS since subparts accessed by different MPI tasks do not overlap. This is not true on the node containing task 0. Pixels computed by other tasks on this node are stored at the same place where they will be received by task 0 since the image is shared. This does not prevent us from making the image HLS. Indeed, as MPC is a thread-based MPI, point to point communications on the same node are realized with `memcpy` and if the source and the destination are identical, which is the case here, this copy is not realized. Thus, sharing the image between MPI tasks on the same node remove the need for intra-node communications.

In order to make these variables HLS, we slightly modified the original source code since the structure containing the scene and the image also contains data that should stay private to each MPI task: buffers for MPI communications and the MPI rank. We split the structure in two parts and the part holding the scene and the image was declared HLS with a pragma. We also added some pragmas **`single`** during scene creation before the rendering phase.

We used a scene containing a high number of objects and textures for a total memory usage of 377MB. The resolution is 4000×4000 which consumes 183MB of memory. The total memory consumption due to the scene is 560MB. On an 8-core node, the memory gain should be around $7 \times 560 = 3920$MB. We computed around 5000 frames.

Table IV compares the memory consumption of Tachyon with MPC, MPC with the HLS mechanism enabled and Open MPI. Conclusions are again similar. MPC consumes less memory than Open MPI and we well observe the expected memory gain. However we also observed in this case an improved execution time due to a reduction of intra node communications on the node containing the MPI task of rank 0. Although this reduction only concerns one node, it still improves the execution time since this is the most loaded node: task 0 needs to receive messages from all the other MPI tasks.

VI. Related Work

For regular process-based MPI runtimes where each MPI task is a process, one can share memory manually between MPI tasks by creating a shared memory segment and allocating variables inside this memory area. This is a complex task which involves a significant rewrite of user code to declare, allocate and access data shared using this technique. A recent proposal in the MPI Forum simplifies the creation of shared memory segments between MPI tasks. It extends the one-sided communications with shared memory windows that can be accessed with regular load and store operations instead of calls to `MPI_Put` and `MPI_Get` for MPI tasks on the same node [14]. The mechanism we propose allows for further simplifying and automating this process for MPI application developers. It requires less code modification, the scope of a variable can be easily chosen and a variable can be efficiently updated thanks to the **`single`** directive. However it is less flexible since it is tailored to the common case of variables that could be shared by all MPI tasks of the application.

Sharing memory between MPI tasks using a shared memory segment is also used by MPI runtime developers to efficiently implement the MPI API. This technique enables process based MPI runtimes to use `memcpy` to implement point to point communications *e.g.*, through buffers or message queues allocated in the shared address space [15]. Collective operations can also be optimized using shared memory algorithms [16], [17].

The HLS mechanism proposes to share variables that were previously private to each execution flow. The reverse operation is called variable privatization. In multithreaded applications, global variables are shared by default and can be privatized using the thread local storage (TLS) mechanism [7]. In OpenMP, global variables can be privatized using the `threadprivate` construct [5]. This directive is often implemented using the TLS mechanism [18]. In thread-based MPI implementations such as MPC [3] or AMPI [19], global variables are by default shared between all MPI tasks on the same node. To be compliant with the MPI standard, these variables need to be privatized. In AMPI, global variables in Fortran are privatized with a set of source-to-source tools based on the Photran [20] plugins. Global variables are packed together in a module and functions are modified to accept this new module as a parameter if needed. In MPC, global variables are privatized using the TLS mechanism. These two different approaches to privatize global variables in thread-based MPI have been compared [21]. For hybrid MPI plus OpenMP code running on a thread-based MPI, it may not be possible to share variables between OpenMP threads if they have been privatized per MPI task. Indeed, if both the OpenMP implementation and the MPI implementation use the TLS mechanism to privatize variables, variables shared between OpenMP threads and private per MPI tasks cannot be distinguished from variables private per OpenMP thread and per MPI tasks. To deal with this problem, an extension to the TLS mechanism is proposed to handle two levels of variables with different privacy attribute: one for MPI and the other for OpenMP [22]. The implementation of the HLS mechanism is based on this extended TLS technique and thus it is compatible with hybrid MPI/OpenMP applications.

Another approach to reduce data duplication of HPC applications is the SBLLmalloc [23] library which automatically merges identical virtual operating system pages of MPI tasks on the same node to the same physical page [23]. SBLLmalloc periodically checks for identical pages, merges them and marks them as read only. When a write occurs, a fault handler unmerges the pages. This technique is fully automatic and does not require the application developer to mark identical data. However, it incurs overhead when scanning for identical pages to be merged and when handling fault to duplicate previously shared pages that have been modified. Moreover it only works at the granularity of a page. Finally, merging pages to the same NUMA node could reduce performances when bandwidth or latency is crucial to the application performance. Our approach does not suffer from this problem since the user can specify if data can be shared across the node or only inside a NUMA node.

Distributed Shared Memory (DSM) systems allow execution flows on different nodes to share the same view of memory [24]. HLS variables are distinct from a DSM as variables are shared only locally on the same node. MPI tasks on different nodes may see distinct values of the same HLS variable as no coherency is kept across nodes. Moreover, contrary to DSM systems, accessing an HLS variable has very little overhead since it does not cause a network communication: all data are kept on all nodes.

VII. Conclusion and Future Work

This article presents HLS, an extension to the MPI parallel programming model that enable to share common variables between MPI tasks on the same node to reduce memory consumption of HPC applications. HLS is a set of directives that preserves the original parallel semantics of the code and is compatible with C, C++ and Fortran languages and the OpenMP programming model. This new mechanism is implemented inside a state-of-the-art MPI 1.3 compliant runtime called MPC [3], [4]. Experiments show that the HLS mechanism can effectively reduce memory consumption of HPC applications. Moreover, by reducing data duplication in the shared cache of modern multicores, the HLS mechanism can also improve performances of memory intensive applications.

A possible extension to this work is to automatically detect variables that can use HLS. One could retrieve during one execution of the code, all memory accesses to global variables augmented with the synchronizations induced by the MPI calls. Efficient algorithms based on the formal definition given in section III could then be used to detect variables that can use HLS without additional synchronizations and to detect where to add synchronizations for the others.

Acknowledgment

This work has been realized in the Exascale Computing Research lab, thanks to the support of CEA, GENCI, Intel and UVSQ.

References

[1] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/openMP parallel programming on clusters of multi-core SMP nodes," in *International Conference on Parallel, Distributed and Network-Based Processing*, 2009.

[2] M. Jowkar, C. Cavazzoni, G. Goumas, and X. Guo, "Report on approaches to petascaling," PRACE, Tech. Rep. [Online]. http://www.prace-project.eu/documents/public-deliverables/d6-4.pdf

[3] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines," in *Europar*, 2008.

[4] P. Carribault, M. Pérache, and H. Jourdren, "Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC," in *International Workshop on OpenMP*, 2010.

[5] OpenMP Architectural Board, "OpenMP Application Program Interface (version 3.0)," 2008.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[7] U. Drepper, "ELF Handling for Thread-Local Storage," 2005. [Online]. http://www.akkadia.org/drepper/tls.pdf

[8] G. Antoniu, L. Bougé, and R. Namyst, "An efficient and transparent thread migration scheme in the PM2 runtime system," in *3rd Workshop on Runtime Systems for Parallel Programming*, 1999.

[9] M. Wolff, S. Jaouen, and H. Jourdren, "High-order dimensionally split Lagrange-remap schemes for ideal magnetohydrodynamics," in *Numerical Models for Controlled Fusion*, 2009.

[10] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, 2005.

[11] Prace project. [Online]. http://www.prace-project.eu/

[12] J. E. Stone, "An efficient library for parallel ray tracing and animation," Master's thesis, Computer Science Department, University of Missouri-Rolla, 1998.

[13] E. Gabriel *et al*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European PVM/MPI Users' Group Meeting*, 2004.

[14] J. Dinan and T. Hoefler, 2011. [Online]. https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/284

[15] D. Buntinas, G. Mercier, and W. Gropp, "Data transfers between processes in an SMP system: Performance study and application to MPI," in *International Conference on Parallel Processing*, 2006.

[16] V. Tipparaju, J. Nieplocha, and D. K. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," in *International Parallel and Distributed Processing Symposium*, 2003.

[17] R. L. Graham and G. M. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," in *European PVM/MPI Users' Group Meeting*, 2008.

[18] X. Matorell, M. Gonzàlez, A. Duran, J. Balart, R. Ferrer, E. Ayguadé, and J. Labarta, "Techniques supporting `threadprivate` in OpenMP," in *International Parallel and Distributed Processing Symposium*, 2006.

[19] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[20] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. M. Ricker, "Automatic MPI to AMPI Program Transformation using Photran," in *Workshop on Productivity and Performance*, 2010.

[21] G. Zheng, S. Negara, C. L. Mendes, E. R. Rodrigues, and L. V. Kale, "Automatic handling of global variables for multi-threaded mpi programs," University of Illinois at Urbana-Champaign, Tech. Rep. 11-23, 2011.

[22] P. Carribault, M. Pérache, and H. Jourdren, "Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications," in *International Workshop on OpenMP*, 2011.

[23] S. Biswas, B. R. de Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong, "Exploiting data similarity to reduce memory footprints," in *International Parallel and Distributed Processing Symposium*, 2011.

[24] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, 1991.