Improving the memory access locality of hybrid MPI applications

Matthias Diener
University of Illinois at Urbana-Champaign
mdiener@illinois.edu

Sam White
University of Illinois at Urbana-Champaign
white67@illinois.edu

Laxmikant V. Kale
University of Illinois at Urbana-Champaign
kale@illinois.edu

Michael Campbell
University of Illinois at Urbana-Champaign
mtcampbe@illinois.edu

Daniel J. Bodony
University of Illinois at Urbana-Champaign
bodony@illinois.edu

Jonathan B. Freund
University of Illinois at Urbana-Champaign
jbfreund@illinois.edu

ABSTRACT

Maintaining memory access locality is continuing to be a challenge for parallel applications and their runtime environments. By exploiting locality, application performance, resource usage, and performance portability can be improved. The main challenge is to detect and fix memory locality issues for applications that use shared-memory programming models for intra-node parallelization. In this paper, we investigate improving memory access locality of a hybrid MPI+OpenMP application in two different ways, by manually fixing locality issues in its source code and by employing the Adaptive MPI (AMPI) runtime environment. Results show that AMPI can result in similar locality improvements as manual source code changes, leading to substantial performance and scalability gains compared to the unoptimized version and to a pure MPI runtime. Compared to the hybrid MPI+OpenMP baseline, our optimizations improved performance by 1.8x on a single cluster node, and by 1.4x on 32 nodes, with a speedup of 2.4x compared to a pure MPI execution on 32 nodes. In addition to performance, we also evaluate the impact of memory locality on the load balance within a node.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; **Distributed architectures**; • **Software and its engineering** → **Main memory**; **Runtime environments**;

KEYWORDS

Memory access locality, load balancing, MPI, AMPI, OpenMP, hybrid applications

ACM Reference Format:
Matthias Diener, Sam White, Laxmikant V. Kale, Michael Campbell, Daniel J. Bodony, and Jonathan B. Freund. 2017. Improving the memory access locality of hybrid MPI applications. In *Proceedings of EuroMPI/USA '17*. ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/3127024.3127038

1 INTRODUCTION

In recent years, many applications for supercomputers have employed a hybrid parallel programming approach, by using different parallel APIs for inter-node and intra-node parallelism. The goals of such a hybrid parallelization are often to eliminate the need for explicit communication between processes in the same node, to decrease the memory footprint on each core, and to be able to balance load across the cores in the node. This approach, when based on using MPI for internode communication, is often referred to as MPI+X, where MPI is paired with a shared-memory API such as OpenMP for intra-node parallelization [34, 36, 40].

Compared to distributed memory APIs, such as MPI, OpenMP has a potentially more efficient way to share data between tasks, as those tasks can access shared memory directly without the need for explicit communication via function calls, as well as avoiding copying data unnecessarily on the same node. In fact, MPI has itself evolved to include support for shared memory parallelism in the MPI-3.0 standard [30].

However, parallel shared memory APIs introduce unique challenges in memory management regarding the *locality* of data accesses that are not present in MPI. For optimal performance, data accesses should be performed to caches and memory controllers on NUMA architectures that are close to where the task that performs the accesses is running, as these accesses have a lower overhead than those to remote caches or memory controllers. As memory is shared between all threads on the same node in an OpenMP environment, care must be taken to place data close to the threads that use it. This can result in significantly faster data accesses in shared memory architectures [3, 7, 9, 11, 16, 33]. On the other hand, data used by each MPI rank is generally private to the rank [14], such that locality issues have a much lower impact on a single cluster node in general.

This paper describes the analysis and changes made to PlasCom2, a multiphysics simulation application based on MPI+OpenMP, to improve its memory locality and discusses the impact of these changes on load balance. We begin with a brief overview of modern shared memory architectures and the main concepts of memory locality, followed by a locality analysis of the PlasCom2 baseline. We then propose improvements to the memory access behavior of PlasCom2 and evaluate their impact on locality, load balance, and performance. We compare the optimized version of PlasCom2 against the baseline version built on Adaptive MPI (AMPI) [24], an implementation of MPI with support for shared memory between ranks.

Our main contributions are the following:

- We extend a tool, numalize, to help with finding and analyzing memory locality issues in shared memory, adding information about allocation and first-touch locations, as well as physical memory addresses.
- We evaluate locality improvements via changes in the application source code and via the AMPI runtime environment.
- We measure the impact of locality on load balance and performance of a hybrid MPI application.

2 MEMORY ACCESS LOCALITY: BACKGROUND AND MAIN CONCEPTS

This section briefly discusses the main concepts of locality and load balance in shared-memory systems. We also introduce the application that we analyze in this paper, as well as the Charm++ and AMPI runtime systems.

2.1 Overview of modern shared memory architectures

Figure 1 shows an example memory hierarchy of a modern shared memory machine. The example contains four memory controllers, each forming a *NUMA node*, which can access a part of the system memory. Several processing cores are attached to each NUMA node. Furthermore, each core has a private cache, and shares another cache with other cores. In this system, a memory access performed by a core can be serviced by a local cache or memory controller, or a remote one.

In such a NUMA machine, the physical address space is divided among the memory controllers. Decisions have to be made to determine the NUMA node on which to place each memory page, by choosing the physical address for the page. Currently, the most common way this placement is performed is via a *first-touch* policy, where the operating system will place each memory page on the NUMA node from which it was first accessed. Such a first-touch policy is currently the default policy in most modern operating systems, including Linux, Solaris, and Windows. Parallel applications need to take this policy into account when aiming to improve locality. By making sure that the first access to each page is performed by the "correct" thread, an application can cause a placement of pages to NUMA nodes that optimizes locality.

2.2 Memory access locality

To evaluate the memory access locality of a parallel application under a first-touch policy, we can measure the percentage of threads that performed the first access to a page and have the most accesses to that page [12]. When scaling this number with the total number of accesses, we can calculate the per-page and overall locality. Values for locality can vary between 0% (when all accesses are remote) and 100% (when all accesses are local).

Related to locality is the first-touch correctness, which indicates the percentage of pages that were accessed first by the thread that performs most accesses to it, which we call the *correct* thread in this context. As locality, first-touch correctness can vary between 0% and 100%.

Table 1 contains an example of these metrics for an application that consists of three threads and that accesses two pages, A and B. Page A receives multiple accesses from several threads, but is accessed first by thread 1, which performs most accesses to the page. This page has therefore a medium locality of 57%. Page B, on the other hand, is accessed mostly by thread 3, but is accessed first by thread 2, and therefore has a low locality.

2.3 Load balance

Apart from memory locality, load balance also is an important aspect of parallel application performance [31]. In order to maximize utilization of all computational resources, the amount of work by each core in the system should be approximately equal, such that no core is overloaded. Load balance is commonly measured by comparing the number of instructions executed by each core. The amount of load imbalance (λ) can be measured with Equation 1 [31], where L is a vector in which each element contains the load of a task of the application. *Load imbalance* varies between 0% (no imbalance) and ∞, with higher values indicating higher imbalance.

$$\textit{Load imbalance}\ \lambda = \left(\frac{max(L)}{avg(L)} - 1\right) \times 100\% \tag{1}$$

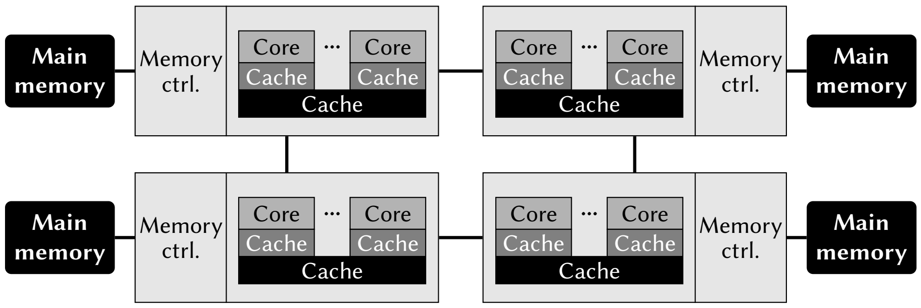

Figure 1: Overview of a parallel shared-memory architecture with four NUMA nodes. Each node consists of a memory controller to which several cores are attached.

Table 1: Example of memory access locality for an application that consists of three threads and accesses two pages.

Page	First touch by thread	Thread 1	Thread 2	Thread 3	Locality	First touch correctness
A	Thread 1	20 accesses	5 accesses	10 accesses	20/(20+5+10) = 57.1%	Correct
B	Thread 2	0 accesses	1 access	10 accesses	1/(0+1+10) = 9.1%	Incorrect
Overall	—	—	—	—	(20+1)/(35+11) = 45.7%	(1/2) = 50%

Load balance is not the main focus of the analysis in this paper, but we will discuss the impact of locality on the load distribution, and, by extension, the performance.

2.4 PlasCom2

PlasCom2 is a next-generation multiphysics simulation application. It is built up from a substrate of modular, reusable components designed to be adapted to modern, heterogeneous architectures. PlasCom2 supports uncertainty quantification, optimization and control, and repeatability through provenance. The parallelization model of PlasCom2 uses domain decomposition with hybrid MPI+X, with X currently as OpenMP. PlasCom2 is written mostly in C++, only the computational kernels are written in Fortran 90 for performance reasons.

The *advect1d* development and testing code exercises several of the key software constructs and features of PlasCom2. It is used to demonstrate PlasCom2 capabilities, and test experimental developments as well as performance. *Advect1d* solves the one-dimensional advection equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \quad \text{on the domain } \vec{x} \in [0,1]^3,\ t > 0 \tag{2}$$

subject to the initial and boundary conditions

$$u(\vec{x},0) = u_0(x), \quad u(0,t) = 0. \tag{3}$$

The general solution is $u(\vec{x},t) = u_0(x-t)$ where we assume that the initial and boundary data are consistent, $u_0(0) = 0$.

2.5 Charm++ and AMPI

Charm++ [1] is an object-oriented parallel programming system based on an asynchronous message driven execution model. Users encapsulate their application's data and computations in entities called *chares*, which are C++ objects. An application written in Charm++ is over-decomposed into these objects so that there are many more objects than physical cores. Chares communicate and synchronize with each other via asynchronous method invocations. Chare objects are assigned to a core by the runtime system, and the user can tell the runtime system to monitor execution and periodically load balance chares by remapping them to cores based on characteristics such as idle time and the communication graph.

Adaptive MPI (AMPI) [24] is an implementation of MPI on top of Charm++ and its adaptive runtime system. It enables use of the main features of Charm++, such as virtualization, load balancing, and online fault tolerance, in MPI codes. AMPI implements MPI ranks as lightweight threads that are migratable among the cores and nodes of a system.

All AMPI programs are MPI programs, the only exception being the use of AMPI's extensions for load balancing and other features. The only restriction on MPI programs being run as AMPI programs is that if they contain mutable global or static variables, those variables must be privatized to the threads. PlasCom2 has been written from the start to avoid use of mutable global and static variables, and so it runs on AMPI unchanged.

AMPI programs can be executed in Charm++'s SMP mode as well, providing shared memory between not only the multiple ranks on a core but across the cores of a NUMA domain. This also allows the runtime to avoid trips through the network for messages sent within a process. By using the SMP mode, AMPI can benefit from the private data semantics of MPI, while allowing the use of shared memory for more efficient operation. In this work, we utilize AMPI in SMP mode to compare its performance in terms of locality and load balance to the hybrid version of the same code and to a traditional MPI runtime environment.

3 ANALYSIS OF THE BASELINE PLASCOM2 VERSION

This section presents the locality analysis of the baseline version of PlasCom2, focusing on the OpenMP runtime.

3.1 The numalize tool

Analysis of the memory access locality of PlasCom2 was performed with the *numalize* tool[1] [12]. Numalize is based on the Intel Pin Dynamic Binary Instrumentation tool[2] [28].

For the analysis presented in this paper, numalize was extended in three major ways. First, numalize was ported from Pin 2 to Pin 3, which required rewriting numalize to not require C++11 features anymore, since Pin 3 requires using its own STL that does not support C++11. This mostly required replacing `std::array` with traditional C-style arrays.

Second, numalize was extended to show data structure names, the source location where they are allocated, and the source location where the first touch is performed. Location information (that is, file name and line number) is determined via Pin's location API, while data structure names are extracted automatically from the application's source code using the location of allocation. All of this information, together with memory addresses, is presented at the granularity of memory pages (4 KByte by default). By default, only data structures that have a size of at least a memory page are considered in order to focus on structures that have a larger impact on the overall memory usage.

Third, numalize now optionally works with physical addresses, obtained from the Linux `/proc` file system, instead of virtual ones. This has the advantage of supporting runtime environments (such

[1]https://github.com/matthiasdiener/numalize
[2]http://pintool.org

Table 2: Example output of the numalize tool, showing statistics for two pages of PlasCom2.

address	alloc.thread	alloc.location	firsttouch.thread	firsttouch.location	struct.name	T0	T1	T2	T3
4324	0	advect.C:387	0	advect.C:555	uBuffer	368,128	0	0	0
4325	0	advect.C:387	0	advect.C:555	uBuffer	26,462	341,870	202	202

as MPI) that are not based on shared memory, but can share data via explicit allocation of shared memory areas. Such areas can be used for optimized communication in a single cluster node, for example [5, 13, 19].

Numalize traces all memory accesses of all tasks at the page granularity, providing information about which part of which data structure was accessed by each thread, and how and where these parts are allocated and accessed for the first time. It creates a `csv` file as output, in which a line per page contains this information. For a typical application, the output file has a compressed size of less than 10 MB.

3.2 Locality analysis with numalize

Table 2 shows a small part of the output of numalize for four PlasCom2 OpenMP threads that access two pages. The table shows the page address, thread ID that allocated the page, source location of the allocation, ID of the thread that performed the first touch, source location of the first touch, data structure name, and number of memory accesses to the page from threads T0–T3. All other statistics, such as locality and first-touch correctness, can be inferred from the values displayed in the table.

In the example, it can be seen that the domain decomposition causes a change in the access behavior, since page 4324 is accessed exclusively by thread 0, while page 4325 is accessed mostly by thread 1. However, both pages are accessed first by thread 0, which determines on which NUMA node both pages are located. Locality is 100% for page 4324, and 7.2% for page 4325.

We measured the overall locality of the baseline version of PlasCom2 for an execution with 1 MPI rank and 8 OpenMP threads. The results indicate a very low locality, with only 31.6% of memory accesses going to local NUMA nodes. 29.9% of the memory pages have a first-touch by the wrong thread. These values show that there is a large potential for locality improvements in the baseline hybrid version of PlasCom2.

4 IMPROVEMENTS TO THE BASELINE VERSION OF PLASCOM2

Analysis of the numalize results revealed a major issue in how most data structures are allocated in the baseline version of PlasCom2. Since the front-end code of PlasCom2 is written in C++, PlasCom2 makes heavy use of `std::vector` to allocate memory. An example of the allocation is shown in Figure 2 for the uBuffer data structure. Importantly, the `resize()` function in line 3 accesses all items in uBuffer, initializing them to 0. As `resize()` is executed by a single thread, the master thread in this example, the complete structure will therefore be first accessed by that thread and placed on the NUMA node where the thread is executing. In this section, we discuss two possible solutions to this issue that are based on parallel initialization of data structures.

4.1 Parallel initialization with manual first touch

Several options to improve this behavior were tested with numalize. Allocating the vector with
`std::vector<double> myBuffer(numPoints);` also results in an initialization to 0. A possible solution to this issue is presented in Figure 3. In this manual *parallel initialization*, first memory for the vector is allocated with the `reserve()` function, which allocates memory, but does not initialize it and which does not update internal vector data structures (such as its size). This newly allocated memory is then initialized in parallel, by each thread, in the parallel loops in lines 5–17. These loops use the same structure in which points will be updated in the normal computation.

Finally, in line 18, the `resize()` function is called to update the vector's internal data structures. Since the vector is resized to the same size that memory was allocated in the `reserve()` function, no new memory will be allocated in this call, and the existing allocation will be reused. Furthermore, since the first access was already performed in the parallel initialization loop, the subsequent initialization in the `resize()` function does not affect the placement of pages to NUMA nodes anymore.

This initialization method does not conform to the C++ standard, as vector elements beyond its current `size()` are accessed. In tests with major C++ compilers (GCC's g++, Intel's icpc, and LLVM clang++), no problem was detected, and the code behaves as expected. However, such a method can be a source of hard-to-debug errors in other compilers or runtime environments. In comparison to manual C-style memory allocation with functions such as `malloc()`, this code has the advantage of maintaining the convenience of using C++ style classes that support STL, as well as removing the need for manual memory management with `malloc()` and `free()`.

4.2 Parallel initialization with a custom C++ memory allocator

Another alternative is to use a custom C++ memory allocator to perform a similar first-touch behavior. For instance, the NUMA-aware memory allocator [21] for the task-based HPX parallel API [25] provides such an allocator, in which each task performs a first-touch

```
1 std::vector<double> uBuffer;
2 // [...]
3 uBuffer.resize(numPoints);
```

Figure 2: Memory allocation in the baseline hybrid version of PlasCom2.

```
1  std::vector<double> uBuffer;
2  // [...]
3  uBuffer.reserve(numPoints); // allocate memory, but do not initialize it
4  // [...]
5  #pragma omp parallel // perform parallel initialization
6  {
7  for(size_t iZ = iStartZ; iZ <= iEndZ; iZ++){
8          size_t zIndex = iZ*xyPlane;
9          for(size_t iY = iStartY; iY <= iEndY; iY++){
10                 size_t yzIndex = zIndex + iY*numX;
11                 for(size_t iX = iStartX; iX <= iEndX; iX++){
12                         size_t xyzIndex = yzIndex + iX;
13                         uBuffer[xyzIndex] = xyzIndex; // perform first touch
14                 }
15         }
16 }
17 }
18 uBuffer.resize(numPoints); // clear buffer again and set its size()
```

Figure 3: Manual memory allocation in optimized version of PlasCom2.

```
std::vector<double, numa_allocator>
     uBuffer(numPoints);
// [...]
class numa_allocator {
// [...]
double* allocate(size_type n, void *hint=0)
{
        double* m = std::allocator<double>::
                    allocate(n, hint);

#pragma omp parallel
        {
        // same code as in Figure 3
        }

        return m;
}
// [...]
}
```

Figure 4: C++ allocator-based memory allocation in optimized version of PlasCom2.

to the data it will access. However, such allocators are rare in the case of OpenMP, as care must be taken that the correct parallel first-touch is applied to each vector.

Figure 4 shows such an implementation for PlasCom2. Here, each vector is allocated with a custom `numa_allocator`, which first allocates memory using C++ standard allocation (without touching the allocated memory), and then performing a similar parallel initialization as presented for the manual allocation in Figure 3 (lines 7–16). As mentioned, the main challenge for such a solution is the need to adapt the allocation strategy for different vectors, if they do not have the same access pattern. For example, the vector indices that each thread accesses might be different. In these cases, it may be necessary to provide different allocator classes for different vectors or specialize them in another way.

4.3 Summary

We presented two ways to improve the first-touch behavior of hybrid applications based on the parallel initialization paradigm. As a side effect of these methods, the allocation of data structures itself is now faster since it is executed in parallel. All the main data structures of PlasCom2 were treated with the two methods described in this section, and we will compare their impact on locality in the next section.

5 ANALYSIS OF IMPROVED VERSIONS

This section presents the locality and load balance gains of the improved versions of PlasCom2, as well as the AMPI version.

5.1 Memory locality

The locality results of the optimized version of PlasCom2 measured with numalize are shown in Figures 5 and 6, as well as Table 3. The results indicate a significant improvement of locality in our two optimized versions. Only 1.2%–1.4% of pages are accessed first by an incorrect thread, while access locality is improved from 32% to 83%. The incorrectly touched pages mostly correspond to shared libraries and communication buffers, on which applications have little influence. As expected, both optimized versions have almost identical locality results.

The AMPI version of PlasCom2 shows a locality and correctness of the first-touch behavior that is comparable to our optimized hybrid versions. This indicates that the MPI semantics of private data translate well to AMPI, even when using its SMP mode. As expected, the MPI runtime (mpich 3.1.3) results also show a high

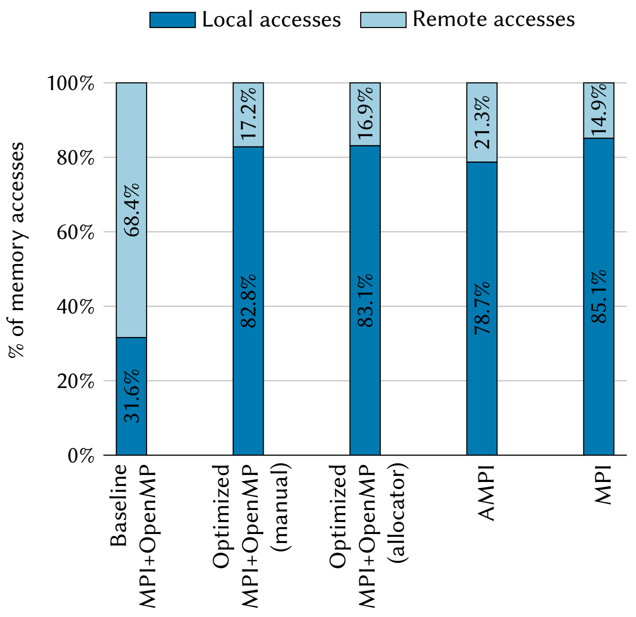

Figure 5: Number of remote vs. number of local accesses of PlasCom2 using different runtimes.

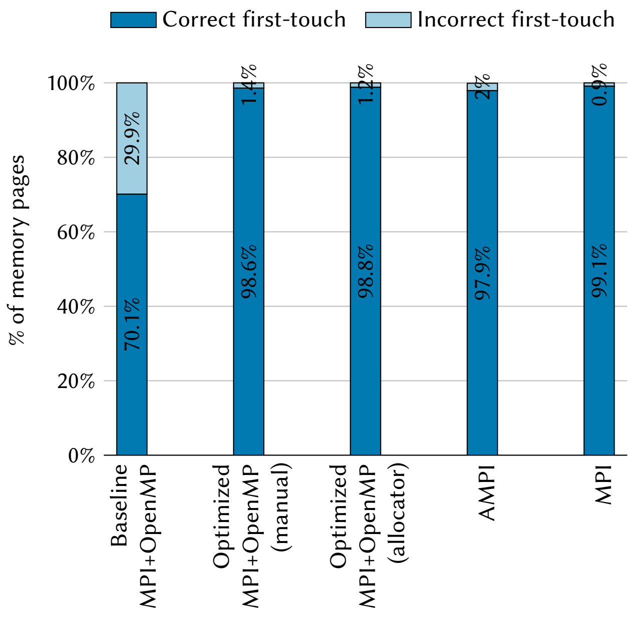

Figure 6: Correct vs. incorrect first touch of PlasCom2 using different runtimes.

locality of 85%, with shared pages mostly related to libraries and communication buffers.

5.2 Load balance

The impact of the locality improvements on the load balance is illustrated in Figure 7. In the figure, we show the number of instructions each of the 8 tasks executed during a complete run of the PlasCom2 versions, measured with the `perf` tool [10]. Table 4 shows the values of the load imbalance metric, calculated with Equation 1. We only show the load imbalance of the manually optimized hybrid version, not the one using the C++ allocator, since results were almost identical. The load imbalance was reduced from 20.6% to 6.7% between the baseline and optimized hybrid versions. The variance in the number of executed instructions per thread was reduced by a factor of 10. The AMPI and MPI versions are the least and most imbalanced, respectively.

The results for the hybrid versions indicate that improving locality can also improve load balance of a parallel application. The reason for this improvement is that tasks are (busy-) waiting for less time, and they are less starving for data.

Table 3: Overall locality for PlasCom2 with 8 threads.

Version	Locality	%pages w/ incorr. first t.
Baseline MPI+OpenMP	31.6%	29.9%
Optimized MPI+OpenMP (manual)	82.8%	1.4%
Optimized MPI+OpenMP (allocator)	83.1%	1.2%
AMPI	78.7%	2.0%
MPI	85.1%	0.9%

Table 4: Load balance metrics for PlasCom2 with 8 tasks.

Version	Load imbalance λ	Variance
Baseline MPI+OpenMP	20.6%	4.2×10^{18}
Optimized MPI+OpenMP	6.7%	2.9×10^{17}
AMPI	3.4%	1.2×10^{17}
MPI	38.2%	1.5×10^{19}

6 PERFORMANCE EVALUATION

To evaluate the performance impact of our locality and load balance improvements in PlasCom2, we perform a series of experiments with single and multiple nodes in a cluster system.

Table 5: Configuration of a single cluster node.

Property	Value
Processor	2× Intel Xeon E5-2680 v3 (Haswell), 12 cores, 2.5 GHz
Caches	32 KByte+32 KByte L1, 256 KByte L2 per core, 30 MByte L3 per processor
Memory	2 NUMA nodes, 64 GByte DDR4-1600 main memory, 4 KByte page size
Software	CentOS 6.8, Linux 2.6.32, gcc 4.9.2, MPICH 3.1.3, Charm++/AMPI 6.7.1
Network	Infiniband interconnection

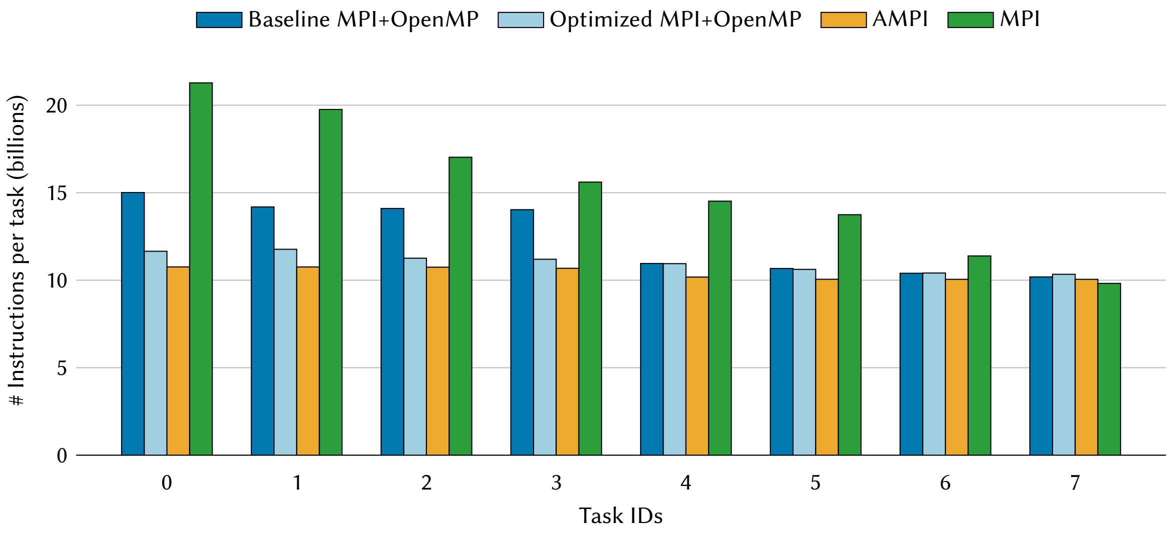

Figure 7: Comparison of load balance for PlasCom2 with 8 tasks. For each task, the number of executed instructions is presented.

6.1 Methodology of the performance experiments

The configuration of the cluster nodes is shown in Table 5. Each cluster node consists of two Intel Xeon Haswell processors, forming a NUMA node each, with 12 cores on each processor. SMT was disabled for the experiments. As each cluster node has 24 cores in total, we run with 24 tasks (MPI ranks or threads) on each node. For the hybrid MPI+OpenMP versions, we execute one MPI rank per cluster node, and 24 threads on each node. Nodes are interconnected with Infiniband.

Four versions of PlasCom2 were compared in the experiments: hybrid (MPI+OpenMP) baseline and manually optimized, MPI-only executed with AMPI 6.7.1 [24] in SMP mode, and MPI-only executed with MPICH 3.1.3 [20]. Results of the hybrid version optimized using the C++ allocator are not shown, since they are almost identical to the manually optimized version. All versions were compiled with gcc 4.9.2 with the -O2 optimization level.

The input problem for each version of PlasCom2 was the same. Our problem size consists of 40 million points per cluster node, resulting in a memory usage of approximately 8 GByte per node. For the experiments with multiple cluster nodes, we perform a weak scaling experiment, in order to maintain a significant main memory usage on each node. We executed each experiment 10 times, and show the average execution time.

6.2 Single cluster node results

Figure 8 shows the execution time results of PlasCom2 running on a single cluster node with two NUMA nodes. We can see a speedup of 1.7× for the optimized hybrid version compared to the baseline, showing the high impact locality and load balance have on the overall application performance. Running with AMPI results in a comparable but slightly faster execution compared to the optimized implementation, with a speedup of 1.8×. Even the pure MPI version

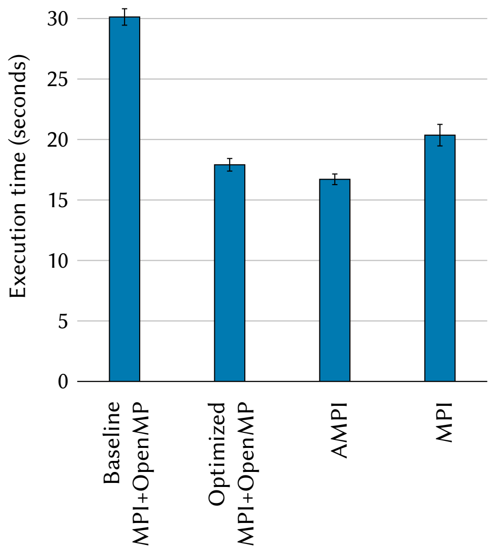

Figure 8: Performance results of PlasCom2 on a single cluster node.

is substantially faster than the hybrid baseline, with a speedup of 1.5×, as it does not suffer from locality issues. This result indicates that locality is more important for performance than load balance in this instance.

6.3 Results on multiple cluster nodes

For the experiments with multiple cluster nodes, we increase the number of cluster nodes from 1 to 32, maintaining 24 tasks (MPI ranks or OpenMP/AMPI threads) per node, with the same configuration

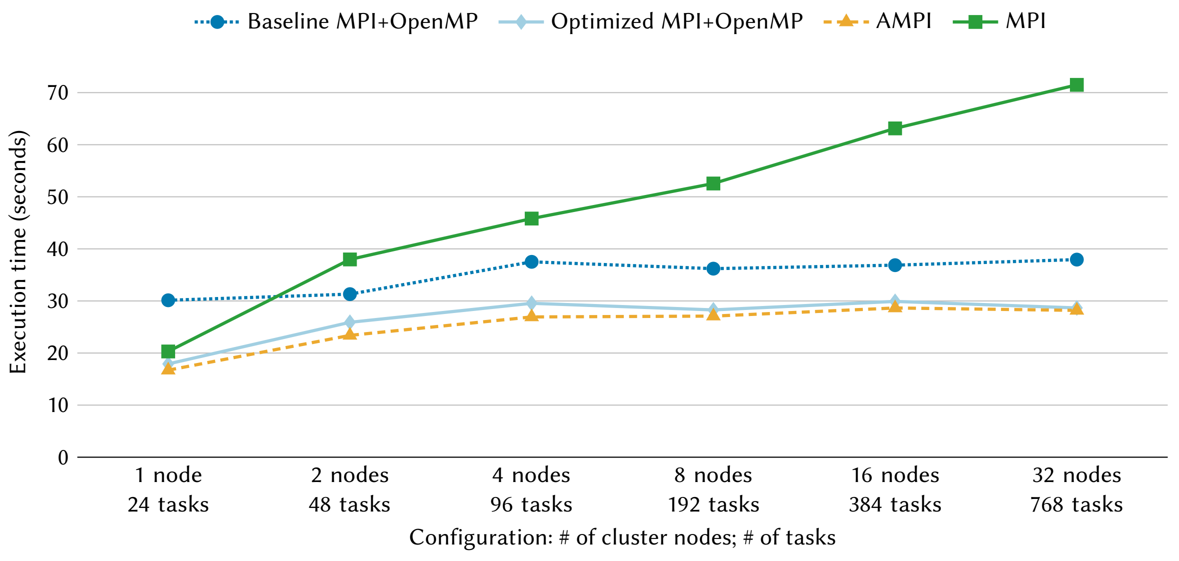

Figure 9: Weak scaling performance results of PlasCom2. The x-axis shows the number of cluster nodes and total number of tasks (MPI ranks or threads).

as before. We perform a weak scaling experiment, that is, the input size is increased linearly with the number of nodes, in order to ensure that the main memory gets used on all nodes. Ideally, the total execution time would remain constant in this experiment for all node counts.

Figure 9 shows the results of this experiment. All implementations that use shared memory scale well up to 32 nodes, only MPI results in increasing execution times with higher node counts, which shows the importance of using shared memory for scaling. Even when running with only 2 nodes, the hybrid baseline is faster then the pure MPI version, reversing the result from the single-node case. The performance difference between the baseline and optimized MPI+OpenMP implementation remains comparable to the single node case, with a speedup of 1.4× on 32 nodes. For 32 nodes, AMPI and the optimized hybrid version show speedups of 2.4× compared to the pure MPI execution. Similarly as in the single-node experiment, the AMPI runtime results in an execution time that is slightly better than our manually optimized version in all cases.

6.4 Summary of performance results

This section has shown that improved memory access locality and load balance can substantially improve performance even in a cluster system. Our manually optimized hybrid version, as well as running with the AMPI runtime environment, are the fastest versions of PlasCom2, showing a significantly better scaling than the pure MPI version, and a lower absolute execution time than the baseline hybrid version. By initializing data structures in parallel, we were able to improve locality and load balance significantly, resulting in substantial performance improvements on shared memory machines.

7 RELATED WORK

The architectural trend toward greater numbers of cores per shared-memory node has exposed greater opportunities for light-weight parallelism within a process. This has led to a rise in hybrid 'MPI+X' programming, where the *X* is a shared-memory programming model such as OpenMP [36]. There has been extensive work on the interoperation of MPI libraries with shared-memory runtimes and their use in hybrid applications [4, 6, 39, 42]. The MPI standard has also grown to include shared memory programming itself [22, 23, 30].

In this work we focus on locality as it relates to load balance, and on the work required to maintain locality in a hybrid model. We use AMPI [24] as our MPI implementation, but others such as MPC [32, 35] and HMPI [15] have similar support for shared memory. Other programming models for cluster architectures, such as those based on the Partitioned Global Address Space (PGAS) model, can also benefit from optimizations to memory access locality, resulting in significant performance gains [3, 21].

Several prior tools can help with detecting and fixing memory access issues on NUMA machines. Generic tools to evaluate parallel application performance, such as Intel's VTune [37], provide summary statistics about the memory access behavior, without clearly identifying sources of potential inefficiencies. More specific tools are therefore required to improve memory locality. Previous proposals that target NUMA architectures, such as MemProf [26], Memphis [29], Liu et al.'s NUMA extension [27] to the HPCToolkit [2], and MemAxes [18], only provide incomplete information that is based on sampling of memory accesses. Sampling can help to understand hot spots in the code, but might lead to incorrect conclusions if it misses important events, such as long-latency loads. Most importantly, sampling usually can not detect the first access to data structures or memory pages, leading to a lack of information regarding the first-touch behavior of an application. The numalize tool [12] discussed in Section 3 has a high one-time overhead, but presents detailed results and detects which thread performs the first access to each page.

Recently, migrating memory pages to improve memory locality during the execution of a parallel application has received renewed attention. Several such mechanisms have been proposed, operating on the hardware level [7, 8, 41], compiler-level [33, 38], or OS-level [9, 11, 17]. These mechanisms do not require changes to the application to improve locality, but can cause a significant runtime overhead that limits their gains compared to the manual changes applied in this paper.

8 CONCLUSIONS

The locality of memory accesses has a high impact on the performance of parallel applications even when executing on cluster systems. For this reason, application developers and users need to be aware of locality issues in their chosen parallel APIs in order to maximize resource utilization. In this paper, we analyze locality issues in a common hybrid parallelization model consisting of MPI and OpenMP. Optimizing the memory locality, via manual changes in the way memory is allocated and accessed first, or by using the AMPI runtime for MPI, results in significant locality and performance improvements, even in a multi-node execution. Compared to the baseline, performance was increased by 1.7× on a single node, and 1.4× on 32 nodes.

Results from the manual improvements and AMPI were very similar, however, both solutions have different tradeoffs. With AMPI, there is no need for hybrid parallelization to achieve the resource utilization of MPI+OpenMP, but users are required to use AMPI and forgo the use of global variables. A custom C++ memory allocator provides the most portable solution, but demands significant developer effort for the implementation. In both cases, scaling was much better than when running a pure MPI implementation, indicating that using shared memory is a key to improving performance of MPI applications. In addition to the performance improvements, we have shown the impact of memory locality on load balance. A higher locality resulted in a lower stall time due to memory accesses, leading to a fairer resource usage.

For the future, we intend to analyze different applications and other parallelization APIs, including those based on PGAS. Furthermore, we will evaluate the impact of different page sizes on locality and performance.

ACKNOWLEDGMENTS

This work is supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice *(SC)*. https://doi.org/10.1109/SC.2014.58

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701. https://doi.org/10.1002/cpe.1553

[3] Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. 2016. Exploiting Hierarchical Locality in Deep Parallel Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 1–25. https://doi.org/10.1145/2897783

[4] Eduard Ayguade, Marc Gonzalez, Xavier Martorell, and Gabriele Jost. 2004. Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International.* https://doi.org/10.1109/IPDPS.2004.1302905

[5] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. 2009. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In *International Conference on Parallel Processing (ICPP).* 462–469. https://doi.org/10.1109/ICPP.2009.22

[6] Julita Corbalan, Alejandro Duran, and Jesus Labarta. 2004. Dynamic load balancing of MPI+OpenMP applications. In *Parallel Processing, 2004. ICPP 2004. International Conference on.* IEEE, 195–202.

[7] Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, and Philippe O. A. Navaux. 2014. Optimizing Memory Locality Using a Locality-Aware Page Table. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* 198–205. https://doi.org/10.1109/SBAC-PAD.2014.22

[8] Eduardo H. M. Cruz, Matthias Diener, Laercio L. Pilla, and Philippe O. A. Navaux. 2016. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 3 (2016), 1–25. https://doi.org/10.1145/2975587

[9] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 381–393. https://doi.org/10.1145/2451116.2451157

[10] Arnaldo Carvalho de Melo. 2010. The New Linux 'perf' Tools. In *Linux Kongress.*

[11] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT).* 277–288. https://doi.org/10.1145/2628071.2628085

[12] Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O. A. Navaux. 2015. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. *Performance Evaluation* 88-89, June (2015), 18–36. https://doi.org/10.1016/j.peva.2015.03.001

[13] Per Ekman, Philip Mucci, Chris Parrott, and Bill Brantley. 2005. *Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: An MPICH/MPICH2 Case Study.* Technical Report.

[14] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* 1–11. https://doi.org/10.1145/2503210.2503294

[15] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 18.

[16] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of memory management on modern NUMA systems. *Commun. ACM* 58, 12 (2015), 59–66. https://doi.org/10.1145/2814328

[17] Ilaria Di Gennaro, Alessandro Pellegrini, and Francesco Quaglia. 2016. OS-based NUMA Optimization: Tackling the Case of Truly Multi-Thread Applications with Non-Partitioned Virtual Page Accesses. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID).* 291–300. https://doi.org/10.1109/CCGrid.2016.91

[18] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting On-Node Memory Access Performance: A Semantic Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* 166–176. https://doi.org/10.1109/SC.2014.19

[19] Brice Goglin and Stéphanie Moreaud. 2013. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. *J. Parallel and Distrib. Comput.* 73, 2 (feb 2013), 176–188. https://doi.org/10.1016/j.jpdc.2012.09.016

[20] William Gropp. 2002. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* https://doi.org/10.1007/3-540-45825-5_5

[21] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. 2016. Closing the Performance Gap with Modern C++. In *ISC High Performance 2016: High Performance Computing.* 18–31. https://doi.org/10.1007/978-3-319-46079-6

[22] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2013. MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing* 95, 12 (2013), 1121–1136.

[23] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian W. Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2012. Leveraging MPI's One-sided Communication Interface for Shared-memory Programming.

In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI’12)*. Springer-Verlag, Berlin, Heidelberg, 132–141. https://doi.org/10.1007/978-3-642-33518-1_18

[24] Chao Huang, Orion Lawlor, and L. V. Kalé. 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*. College Station, Texas, 306–322.

[25] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *International Conference on Partitioned Global Address Space Programming Models (PGAS)*. 6:1—-6:11. https://doi.org/10.1145/2676870.2676883

[26] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX Annual Technical Conference (ATC)*. 53–64.

[27] Xu Liu and John Mellor-Crummey. 2014. A tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 259–272. https://doi.org/10.1145/2555243.2555271

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200. https://doi.org/10.1145/1065010.1065034

[29] Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 87–96. https://doi.org/10.1109/ISPASS.2010.5452060

[30] Message Passing Interface Forum. 2012. *MPI: A Message-Passing Interface Standard (Version 3.0)*. Technical Report.

[31] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Martin Schulz, and Nancy M. Amato. 2012. Quantifying the effectiveness of load balance algorithms. In *ACM International Conference on Supercomputing (ICS)*. 185–194. https://doi.org/10.1145/2304576.2304601

[32] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing (Euro-Par ’08)*. Springer-Verlag, Berlin, Heidelberg, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9

[33] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 369–380. https://doi.org/10.1145/2628071.2628077

[34] Boris V Protopopov and Anthony Skjellum. 2001. A multithreaded message passing interface (MPI) architecture: Performance and program issues. *J. Parallel and Distrib. Comput.* 61, 4 (2001), 449–466. https://doi.org/10.1006/jpdc.2000.1674

[35] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users’ Group Meeting (EuroPVM/MPI 2009)*, Matti Ropo, Jan Westerholm, and Jack Dongarra (Eds.). Lecture Notes in Computer Science, Vol. 5759. Springer Berlin Heidelberg, 94–103. https://doi.org/10.1007/978-3-642-03770-2_16

[36] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. 2009. Hybrid MPI and OpenMP Parallel Programming MPI + OpenMP and other models on clusters of SMP nodes. *17th Euromicro Int. Conf. Parallel, Distrib. Networkbased Process.* d (2009), 427–436. https://doi.org/10.1109/PDP.2009.43

[37] James Reinders. 2005. *VTune performance analyzer essentials*. Intel Press. http://www.openisbn.com/isbn/9780974364957/

[38] Christiane Pousa Ribeiro, Marcio Castro, Jean-François Méhaut, and Alexandre Carissimi. 2010. Improving memory affinity of geophysics applications on NUMA platforms using Minas. In *International Conference on High Performance Computing for Computational Science (VECPAR)*. 279–292. https://doi.org/10.1007/978-3-642-19328-6_27

[39] Lorna Smith and Mark Bull. 2001. Development of Mixed Mode MPI / OpenMP Applications. *Scientific Programming* 9, 2,3 (Aug. 2001), 83–98. http://dl.acm.org/citation.cfm?id=1239928.1239936

[40] Hong Tang and Tao Yang. 2001. Optimizing threaded MPI execution on SMP clusters. *ICS ’01: Proceedings of the 15th international conference on Supercomputing* (2001), 381–392. https://doi.org/10.1145/377792.377895

[41] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing (JPDC)* 68, 9 (sep 2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[42] Rob F. Van der Wijngaart and Haoqiang Jin. 2003. *NAS Parallel Benchmarks, Multi-Zone Versions*. Technical Report.