Exploiting Data Similarity to Reduce Memory Footprints

Susmit Biswas, Bronis R. de Supinski, Martin Schulz
Lawrence Livermore National Laboratory
Livermore, CA - 94550, USA
Email: {biswas3, bronis, schulzm}@llnl.gov

Diana Franklin, Timothy Sherwood, Frederic T. Chong
Department of Computer Science
University of California, Santa Barbara, USA
Email: {franklin, sherwood, chong}@cs.ucsb.edu

***Abstract*—Memory size has long limited large-scale applications on high-performance computing (HPC) systems. Since compute nodes frequently do not have swap space, physical memory often limits problem sizes. Increasing core counts per chip and power density constraints, which limit the number of DIMMs per node, have exacerbated this problem. Further, DRAM constitutes a significant portion of overall HPC system cost. Therefore, instead of adding more DRAM to the nodes, mechanisms to manage memory usage more efficiently —preferably transparently— could increase effective DRAM capacity and thus the benefit of multicore nodes for HPC systems.**

MPI application processes often exhibit significant data similarity. These data regions occupy multiple physical locations across the individual rank processes within a multicore node and thus offer a potential savings in memory capacity. These regions, primarily residing in heap, are dynamic, which makes them difficult to manage statically.

Our novel memory allocation library, *SBLLmalloc*, automatically identifies identical memory blocks and merges them into a single copy. Our implementation is transparent to the application and does not require any kernel modifications. Overall, we demonstrate that *SBLLmalloc* reduces the memory footprint of a range of MPI applications by 32.03% on average and up to 60.87%. Further, *SBLLmalloc* supports problem sizes for IRS over 21.36% larger than using standard memory management techniques, thus significantly increasing effective system size. Similarly, *SBLLmalloc* requires 43.75% fewer nodes than standard memory management techniques to solve an AMG problem.

I. Motivation

Memory dominates the cost of HPC systems. The density of DRAM components double every 3 years while logic components double every 2 years. Thus, memory size per core in commodity systems is projected to drop dramatically, as Figure 1 illustrates. We expect the budget for an exascale system to be approximately $200M and memory costs will account for about half of that budget [21]. Figure 2 shows that monetary considerations will lead to significantly less main memory relative to compute capability in exascale systems even if we can accelerate memory technology improvements [21]. Thus, we must reduce application memory requirements per core.

Virtual memory and swapping can increase effective memory capacity. However, HPC applications rarely use them due to their significant performance penalty and a trend towards diskless compute nodes in order to increase reliability.

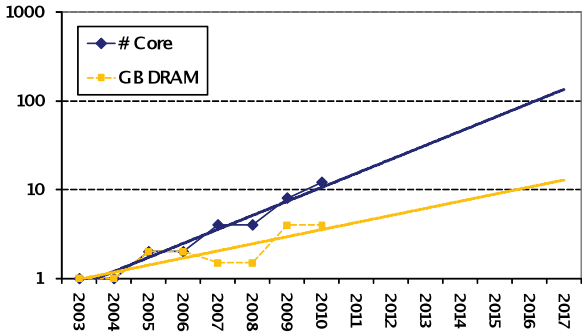


Fig. 1: Memory Capacity Wall

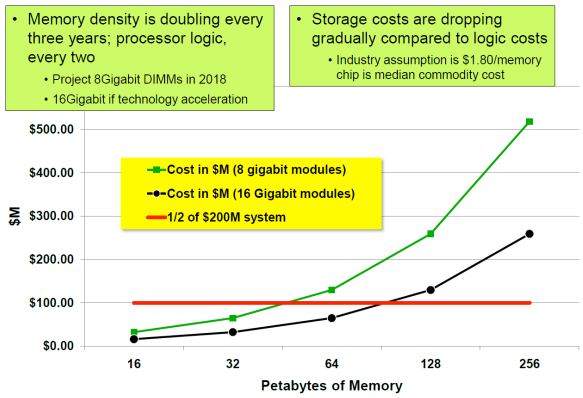


Fig. 2: Cost curve [21]

Prior proposals to reduce memory footprints based on Distributed Shared Memory (DSM) [14, 16] require users to identify common data regions and to share memory explicitly across nodes. These solutions require modifications to identify common data regions in the source code that make an application difficult to port and to maintain. In addition, the system can only benefit from similarities that the programmer can explicitly identify, making it especially difficult to exploit regions that are usually similar but not always. Our studies show that this dynamic similarity is common in MPI programs.

Kernel level changes [7, 10, 13, 22, 23, 24] can reduce the application changes required to leverage data similarity. However, these solutions require more effort from system administrators, which complicates their adoption in production systems. The high performance computing (HPC) community

needs an automated, *user-level* solution that exploits data similarity to reduce memory footprints.

We present *SBLLmalloc*, a *user-level* memory management system that *transparently* identifies identical data regions across tasks in an MPI application and remaps such regions for tasks on the same node to use the same physical memory resource. *SBLLmalloc* traps memory allocation calls and transparently maintains a single copy of identical data in a content-aware fashion using existing system calls.

In this paper we make the following contributions:

- Detailed application studies that show identical data blocks exist across MPI tasks in many applications;
- A user-level memory management library to reduce memory footprints with no OS or application modifications;
- Scaling and overhead results of the library for a range of input sizes for several large-scale applications;
- A demonstration that *SBLLmalloc* enables large problem size executions that are impossible with the default memory allocator due to out-of-memory errors.

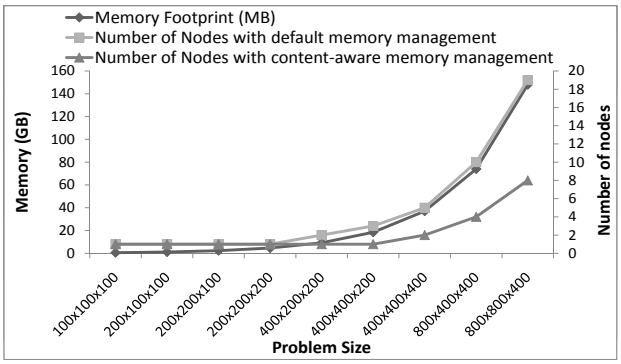
Overall, our system transparently reduces *peak* memory consumption of our test applications by up to 60% (32% on average). More importantly, *SBLLmalloc* supports a 21.36% larger problem size of IRS, an implicit radiation solver application, using the same hardware resources. Further, we can solve a problem of AMG, an Algebraic Multi-Grid solver, that requires 128 nodes with the default allocator, using only 72 nodes with *SBLLmalloc* (i.e., 43.75% fewer nodes).

The paper is organized as follows. Section II motivates our problem by showing the high degree of data similarity in MPI applications. We describe the *SBLLmalloc* implementation of our techniques to leverage this similarity in Section III. We describe our experimental setup in Section IV and present extensive results with *SBLLmalloc* in Section V.

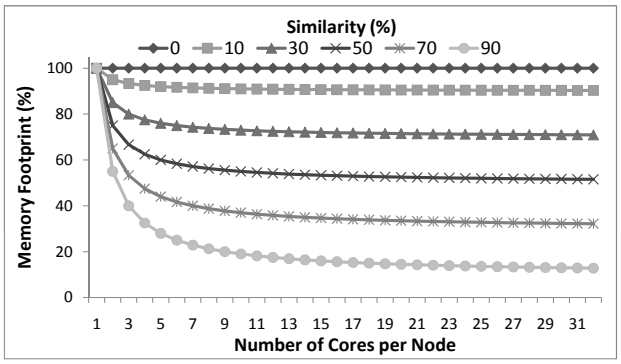
II. Data Similarity in MPI Applications

Memory capacity significantly limits problem sizes, as a simple example demonstrates. As the problem size grows, more nodes are necessary in order to meet the increased memory requirements. Figure 3(a) shows a case study of AMG from the ASC Sequoia benchmark suite [2] on a 128-node system, each node having 8 GB of main memory. The default memory management line shows that as the problem size grows, the number of nodes necessary also grows.

Reducing memory requirements per process is desirable because of the cost of memory and the gap between memory size and computation power. Previous studies on cache architectures [5] found significant data similarity across concurrent, yet independent, application runs. MPI applications composed of SPMD (Single Program Multiple Data) processes more prominently exhibit data similarity. We can leverage this similarity across MPI tasks to reduce the memory footprint in every node significantly. Figure 3(b) shows the reduction in memory footprint that can be attained in the ideal case given different amounts of data similarity. For example, with 70%



(a) Benefits of content-aware memory management



(b) Reduction by exploiting data replication

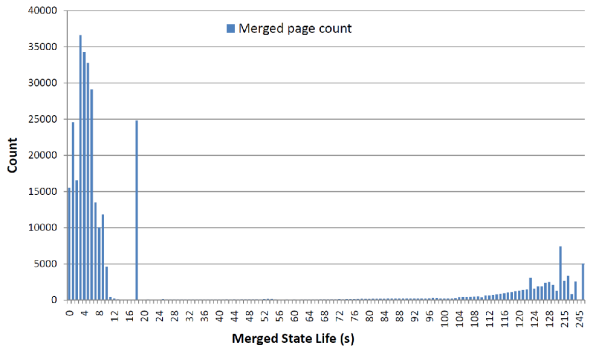
Fig. 3: Content-aware memory management benefits

data similarity, a content-aware memory allocator can reduce the memory footprint by over 60% for an 8-core node. Finally, the content-aware memory management line in Figure 3(a) shows the actual reduction in nodes necessary when using our allocator. For example, a problem that normally requires 19 nodes can be solved with only 8 nodes.

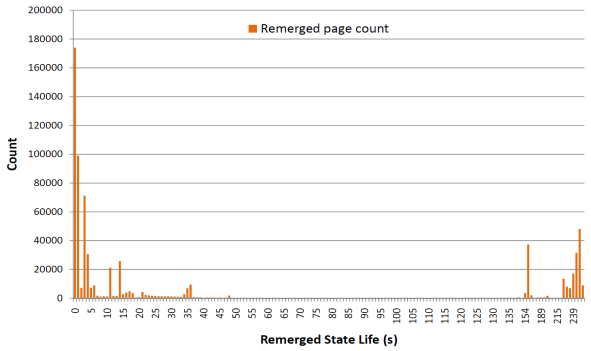
We could reduce the memory footprint by identifying common objects statically and using shared memory. However, this method requires significant programmer effort, which must be repeated for each application in order to support scaling without losing computational power. Further, data similarity is *dynamic*, which further complicates manual solutions and limits the effectiveness of static approaches.

We find that significant dynamic similarity exists in data blocks across MPI tasks. We used Pin [19] to develop a dynamic binary instrumentation tool to analyze the nature of the data similarity. Our results indicate that data similarity primarily occurs in heap-allocated regions. Thus, we use heap management to merge identical data.

Our Pin tool characterizes identical data as *Merged* or *Remerged*. The latter type captures data blocks that were similar after initialization, and so could have been merged (i.e., *Merged* blocks), then diverge due to modifications but again become identical later. Many applications exhibit this recurring data similarity, as Figure 4 shows for AMG. A remerged point indicates that the corresponding page undergoes merge and write phases. A bar of height x at horizontal point y

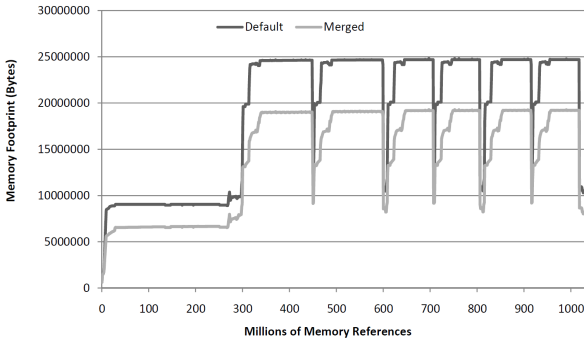


(a) Merged

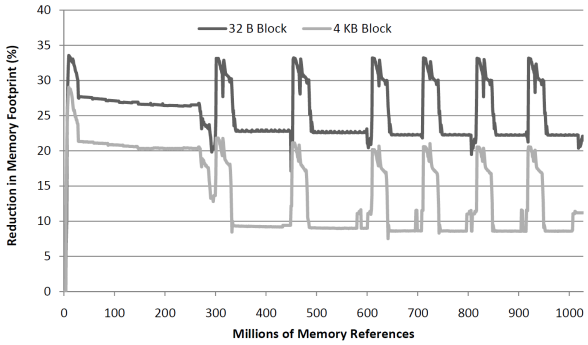


(b) Remerged

Fig. 4: Nature of merged blocks in AMG



(a) Memory footprint over lifetime



(b) Effect of granularity

Fig. 5: Block size analysis of IRS using Pintool

indicates that x blocks stayed merged for y seconds. While similarity is often transient (pages that stay merged for a short duration) and many pages exhibit write-once behavior (pages that are merged for the entire execution), many pages are remerged repeatedly, even periodically. Figure 5(a), in which our tool tracks `malloc/free` calls, clearly shows this phased behavior in IRS.

Our goal is a portable, transparent, high-performance solution. We achieve transparency and portability by designing and developing *SBLLmalloc* as a user-level library that intercepts allocation invocations (`malloc` calls) and manages identical data blocks using shared memory. Since we locate identical data in shared-memory pages, we consider the impact of block size on data similarity. Our experiments demonstrate that sharing data blocks at fine granularity leads to larger memory footprint reductions, as Figure 5(b) shows for IRS. Although we achieve a larger benefit with a larger problem size, the trend remains the same: smaller block sizes capture more similarity. However, our user-level shared data management requires large block sizes (at least page size) to keep overhead manageable. Moreover, we find that smaller sized blocks, e.g., 256KB, hardly increase the total mergeable data (often less than 5%) as most allocated data blocks are large.

III. Exploiting Data Similarity

We designed *SBLLmalloc* to provide a portable, transparent user-level mechanism to exploit data similarity in MPI tasks. Our key challenge was that our goal of a transparent, user-level mechanism severely limited the granularity of merges as well as how and when we identify and merge pages.

```
/* create allocation record */
allocRecord = (AVLTreeData*) CreateAVL();

/* create semaphore */
mutex = sem_open(SEMKEY, ...);

/* open shared file */
sharedFileDescr = shm_open("/shared_region", ...);

/* map system-wide zero page */
zeroPage =(char*)mmap(NULL, PAGE_SIZE, PROT_READ,
    MAP_SHARED, sharedFileDescr, 0);

/* bit-vector to track sharing */
shBV = (int*) mmap(NULL, ... , MAP_SHARED,
    sharedFileDescr, ...);
```

Fig. 6: SBLLmalloc initialization routine

The core components are POSIX locks and shared memory segments, through which we manage page mappings. *SBLLmalloc* uses a shared memory pool that we obtain with `shm_open()` to handle allocations of 4KB (the page size on our experimental systems) or more. Smaller allocations use the default allocator from *glibc*. Figure 6 shows our initialization pseudocode, which we invoke during `MPI_Init`. This phase obtains the shared memory region and initializes key data structures, such as the AVL tree that we use to track allocations.

SBLLmalloc uses system calls `mprotect`, `mmap`, and `mremap` to control access privileges (shared vs. private) and

```
/* allocate */
void* ptr = (void *) mmap(NULL, size,
    ... /* read-only and private */, -1, 0);

/* store allocation record */
AvlInsert(ptr2offset(ptr), size);
```

Fig. 7: SBLLmalloc allocation routine

virtual-to-physical mappings (merging). It moves data between physical pages for remapping with `memcpy`. Mapping changes only the physical location, not the virtual location, so a remap does not affect code correctness. We currently only consider merging full pages since these mechanisms only work on a per-page granularity.

We observe that many dirty pages (initialized data) contain only zeros, so we apply an optimization that recognizes *zero* pages. We map the zero pages in all tasks to a single physical page. Zero page management further reduces memory footprints by merging data regions within MPI tasks as well as across them. Finally, we can use `memset` rather than `memcpy` to reduce copy overhead.

Our library is transparent to applications. It must link to our library, which then intercepts all accesses to `malloc` (or `new`). *SBLLmalloc* provides its benefits by periodically checking for identical data and merging pages. It marks merged (shared) pages as copy-on-write, so a write to a merged page triggers an interrupt, in response to which a handler unmerges the page. Figure 7 shows that *SBLLmalloc* initially allocates pages as read-only and private to the requesting process. We track the allocation in our AVL tree, which is a height-balanced data structure that provides fast searching.

Identifying identical pages is expensive so we limit attempts to specific *merge points* that only compare pages with the same virtual starting address and that have been accessed since the last merge point. Since our goal is to reduce peak memory usage, *SBLLmalloc* monitors node memory usage with internal counters and only initiates the merge process when it exceeds a configurable threshold (*e.g.,* 80%) of the available physical memory. For example, in a node with $12GB$ physical memory, we could set this threshold to $8GB$, assuming $2GB$ is used by system processes. In the merge process, one task moves a private page to the shared space and sets it as *read-only* using the `mprotect` call, as Figure 8 shows. Then other tasks compare their private pages at the same virtual address with the page. If a task's page matches the shared page then we map the shared physical page to the same virtual address of its private page using `mremap`. Since each page is initially read-only, any write to it causes a page fault that invokes the *SBLLmalloc* `SIGSEGV` handling routine that Figure 9 shows. This routine makes the faulting page writable and invokes the merge process if memory usage exceeds the threshold.

We merge pages in bulk to reduce overhead. We classify the pages to be merged in three categories. A zero page indicates that the page contains only zeros. A sharable page is a private page for which there is no shared page in any process at the same virtual address. A mergeable page has identical data as a shared page in another process at the same virtual address. *SBLLmalloc* maintains a queue of pending page addresses for each of these categories. Figure 10 shows the state machine that we use to categorize and to process groups of pages once our interrupt handler observes that memory usage exceeds the threshold. This state machine uses `MergePendingPages` to perform the merging, which uses `mmap`, `mremap`, and `memcpy`, as Figure 11 shows.

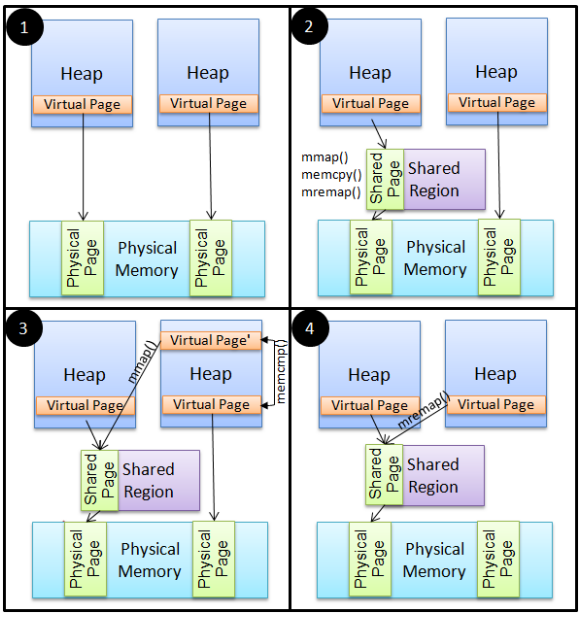


Fig. 8: Page merging process

```
if(permissionFault) {
    /* change permission */
    mprotect(faultAddr, len, ... /* read/write */);

    /* change sharing state */
    UnsetSharingBit(shBV, faultAddr);
}
else
    ...  /* raise exception */

if(... /* memory usage crosses threshold */)
    MergePages();
```

Fig. 9: SBLLmalloc fault handler

Not surprisingly, merge operations account for most of the library's overhead. Figure 12 shows the overhead of different merge operations of several microbenchmarks on an Ubuntu Linux box running on an AMD Phenom 9600 Quad-Core system. Because `mprotect` calls dominate the overhead, we group contiguous pages that require permission changes. We maintain a queue of contiguous pages that should be shared or mapped to the zero page. We apply the same optimization to `mremap` operations. We analyze the overhead of the merge operation in detail in Section V.

IV. Experimental Methodology

We describe our three sets of evaluations in this section. We then describe the benchmark applications that we used.

```
for all pages{
  ... /* map a group of pages in bulk */
  for all pages from the group{
    if(... /* not dirty or already shared */)
      MergePendingPages();
    else if(... /* this page is a zero page */)
      if (... /* last page is not a zero page */)
        MergePendingPages();
      else
        ... /* add to list of
            "to be mapped to zero" pages */
    else if(... /* no other process has
              this page in shared region */)
      if (... /* last page is not a shareable page */)
        MergePendingPages();
      else
        ... /* add to list of shareable pages */
    else if(... /* content of this page matches
              with the shared region */)
      if (... /* last page is not a mergeable page */)
        MergePendingPages();
      else
        ... /* add to list of mergeable pages */
    else
      MergePendingPages();
  }
  ... /* unmap the pages */
}
MergePendingPages();
```

Fig. 10: SBLLmalloc merge routine

```
if(... /* shareable pages */) {
   mmap(...); memcpy(...); mremap(...); }
if(... /* mergeable pages */)
   mremap(...);
if(... /* zero pages */)
   mmap(...);
/* set bit-vectors */
...
```

Fig. 11: Heart of MergePendingPages routine

A. Inspection Experiments

Our first set of evaluations use a dynamic binary instrumentation tool built with the Pin API (*Pin-2.6*) to monitor memory accesses and to compare their contents across MPI tasks. We use this tool to inspect various aspects of data similarity, such as the effect of block size on merge behavior, as explained in Section II. We use this tool on a Dell Optiplex 700 system ($2.8GHz, 1GB$ DRAM, Core Duo 2, CentOS 4.2) along with *mpich-1.2.7* [12] as the underlying MPI implementation.

B. Scaling Experiments

For our other evaluation sets, we use a 128-node cluster with 8 Intel Xeon cores ($2.66GHz$) per node running the Chaos operating system [11] (2.6.18 kernel). Each node has $12GB$ memory. MVAPICH serves as our MPI implementation and we compile all programs with gcc-4.1.2. We collect all statistics using counters internal to the library. For timing results we use the Linux *time* utility. We run the experiments multiple times (typically between 2 and 10) and report the variation along with the average.

Our second evaluation set consists of two types of scaling experiments. Our *problem size scaling* experiments keep the task count constant while changing the input parameters to vary memory requirements. Our *task scaling* experiments change the task count (and, thus core count used) per node.

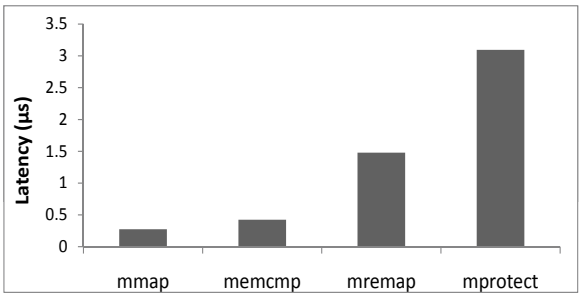


Fig. 12: Overhead of operations

C. Capacity Experiments

Our third evaluation set tests how well the library extends the memory capacity limit. These tests use the same task count as our problem size scaling experiments, but use input sets that require close to $11GB$ of memory per node, leaving $1GB$ for system processes. We perform two types of experiments. The first runs larger problems on the same node count and memory capacity. The second runs the same problem using a smaller node count and memory. Both types show that *SBLLmalloc* enables larger problem sizes using an existing system than the default memory management. The latter studies the capability to solve a problem instead of memory footprint reduction efficiency. We present this evaluation set for only two applications (IRS and AMG) although all applications show this behavior.

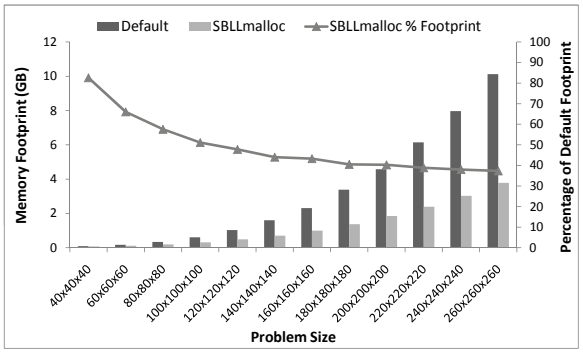
D. Benchmarks

Our study uses four (*AMG, IRS, LAMMPS, UMT*) ASC Sequoia Tier 1 benchmarks [2], seven applications from SPEC MPI2007 [1], DT from the NAS Parallel Benchmarks [4], and two real world applications, ParaDiS and Chombo.

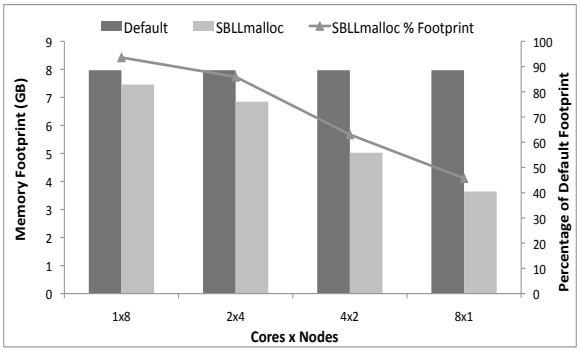
We present results for the SPEC MPI2007 benchmarks that use dynamic memory allocation and can be compiled successfully using GCC C or Fortran compilers. The remaining SPEC MPI benchmarks either do not perform dynamic memory allocation or cannot be compiled using gfortran. DT is the only NAS benchmark that allocates memory dynamically; thus, we do not study other NAS benchmarks. *ParaDiS* [8] computes the plastic strength of materials by tracing the evolution of dislocation lines over time. We experiment with only two data sets because availability of proprietary data sets is limited. *Chombo* [3] provides a distributed infrastructure for parallel calculations such as finite difference methods for solving partial differential equations over block-structured, adaptively refined grids using elliptic or time-dependent methods. We experiment with elliptic and Godunov AMR methods. We provide brief descriptions of our benchmarks in Table I.

	Benchmark	Languages	Description
ASC Sequoia	*AMG*	C	Algebraic Multi-Grid linear system solver for unstructured mesh physics packages
	IRS	C	Implicit Radiation Solver for diffusion equations on a block structured mesh
	LAMMPS	C	Full-system science code targeting classical molecular dynamics problems
	UMT	C, Python and Fortran	Single physics package code for Unstructured-Mesh deterministic radiation Transport
SPEC MPI2007	*104.milc*	C	Quantum Chromodynamics (QCD) application
	107.leslie3d	Fortran	Computational Fluid Dynamics (CFD) application
	122.tachyon	C	Parallel Ray Tracing application
	128.GAPgeofem	C and Fortran	Simulates heat transfer using Finite Element Methods (FEM)
	129.tera_tf	Fortran	3D Eulerian Hydrodynamics application
	132.zeusmp2	C and Fortran	Simulates Computational Fluid Dynamics (CFD)
	137.lu	Fortran	Computational Fluid Dynamics (CFD) application
NPB [4]	*DT*	C	Models data communication with large graphs to evaluate communication throughput
Real Appl.	*ParaDiS* [8]	C	Computes plastic strength of materials by tracing dislocations
	Chombo [3]	C++	Adaptive Mesh Refinement (AMR) package. Experiment with Elliptic and Godunov methods

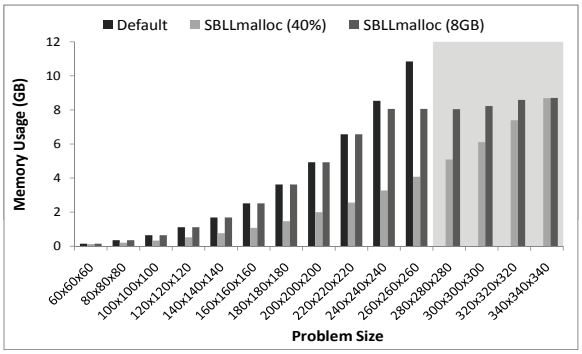
TABLE I: Benchmarks



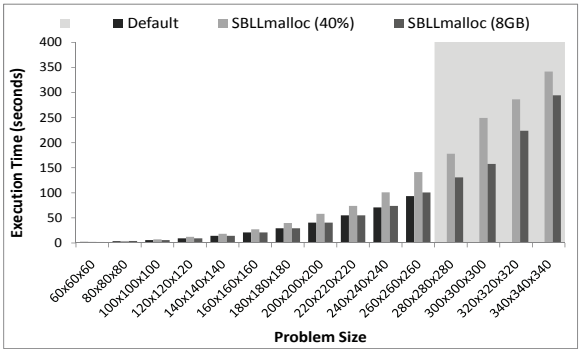
(a) Problem size scaling results



(b) Task scaling



(c) Memory Reduction



(d) Execution Time

Fig. 13: Scaling results and analysis of AMG (shaded region indicates infeasible problem sizes with the default allocator)

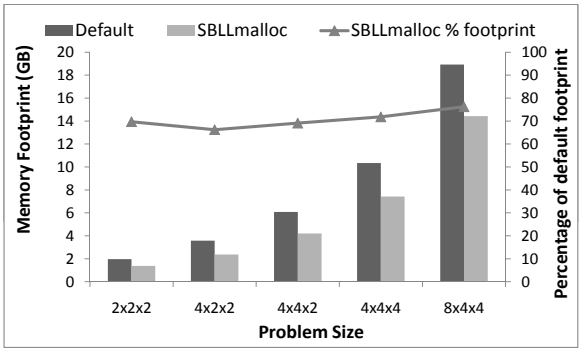
V. Results

In this section, we evaluate *SBLLmalloc* for each application. We present results for the peak memory usage reduction with problem size scaling and with scaling tasks per node as well on the overhead of merging on execution time. We also study the characteristics of merged pages to understand the source of data similarity in three representative applications.

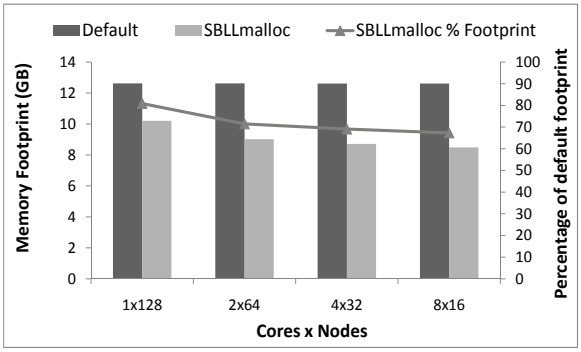
A. Problem Size and Task Scaling

In Figure 13(a) we show the effect of problem size on memory footprint reduction when running the problem on a single node using 8 tasks for AMG. The left vertical axis shows the absolute memory footprint for AMG tasks with and without *SBLLmalloc* (presented with bars), and the right axis shows the memory footprint with *SBLLmalloc* relative to the default glibc malloc (presented with lines). A negative slope of the line indicates that *SBLLmalloc* becomes more effective with larger problem sizes. We observe a similar trend with the task scaling experiments in Figure 13(b) where *SBLLmalloc* shows more benefit with larger task counts per node. With 8 tasks and a large problem size, the resulting memory footprint becomes less than 38% of the original. With one task per node, merging zero pages reduces the memory footprint by 7%.

The overhead in the merge process depends on our trigger threshold. Lowering it causes more frequent merging. To determine merging overhead, we vary the threshold and measure the execution overhead. In Figures 13(c) and 13(d), we show results for two *SBLLmalloc* trigger scenarios. The one identified by *SBLLmalloc* (40%) indicates executions in which we set the memory usage threshold for the merge process to

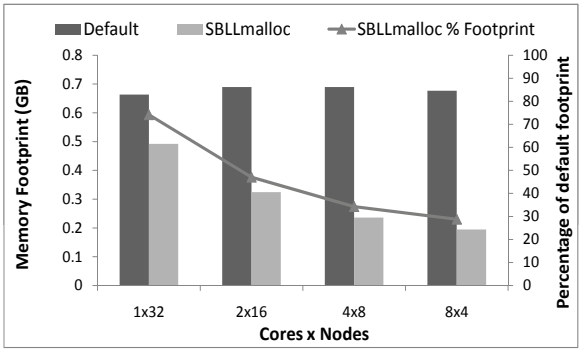


(a) Problem size scaling results

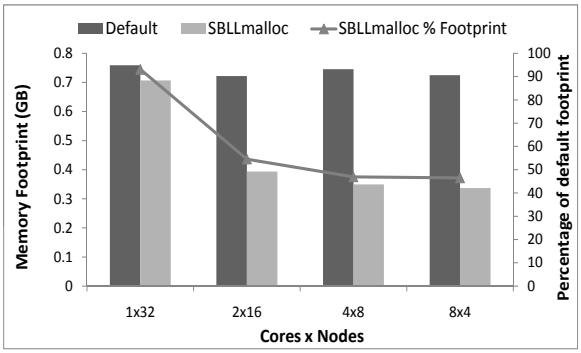


(b) Task scaling results

Fig. 14: LAMMPS scaling results

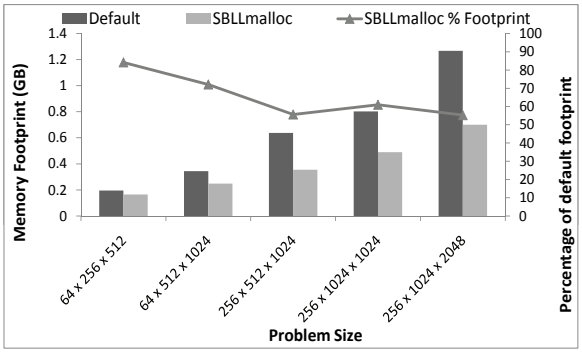


(a) Task scaling results: Form inputs

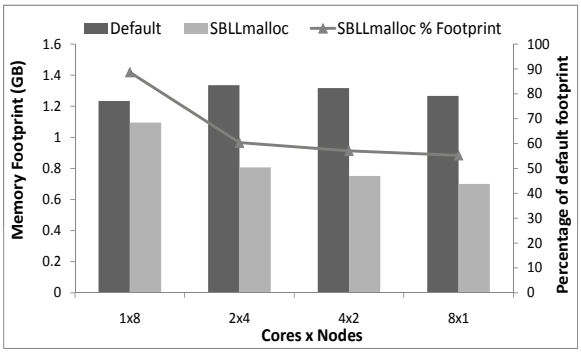


(b) Task scaling results: Screw inputs

Fig. 15: ParaDiS results with SBLLmalloc



(a) Problem size scaling results

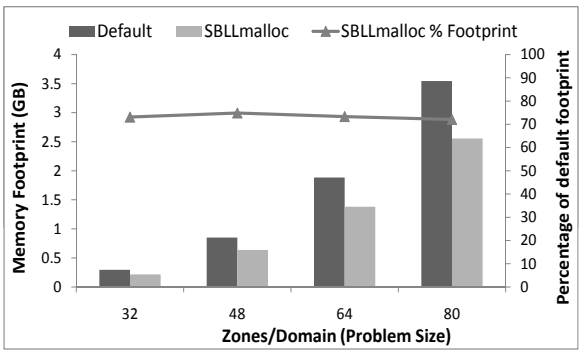


(b) Task scaling results

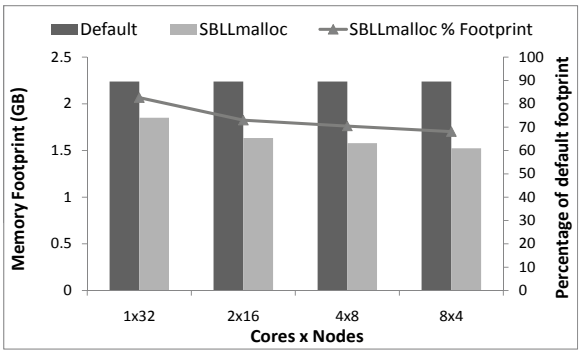
Fig. 16: Chombo results with Godunov solver

40% of the memory used with the default allocator. The other use scenario corresponds to ones in which we set the threshold to 80% of available physical memory. As each node has $12GB$ of memory, leaving $2GB$ for the system services, we set the threshold as $8GB$ in this case. Setting the threshold as high as $8GB$ does not incur any runtime overhead when the memory requirement is less than $8GB$ per node. Any problem larger than $260 \times 260 \times 260$ does not fit in a single node with the default allocator, but problem sizes up to $340 \times 340 \times 340$ fit with *SBLLmalloc*. Thus, *SBLLmalloc* enables running small problems with little or no overhead in a realistic use case, i.e., a fixed and high memory threshold (e.g., $8GB$), while it reduces the memory footprint up to 62.45% for cases in which memory is scarce at the cost of slowing execution by $6.54 - 51.5\%$ as shown by the variable threshold experiments, *SBLLmalloc* (40%).

Memory footprint reductions for LAMMPS degrade slightly with increased problem size, as Figure 14(a) shows. These experiments use 128 MPI tasks, distributed across 16 nodes. Our task scaling experiment, in which we vary the task count per node from 2 to 8 by a factor of two in each step, shows that more tasks per node increases the benefit of *SBLLmalloc*, reducing the memory footprint by 32.5% per node. Increasing the problem size is less likely to increase

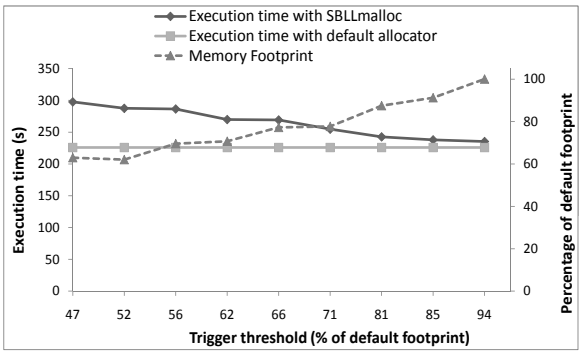


(a) Problem size scaling results

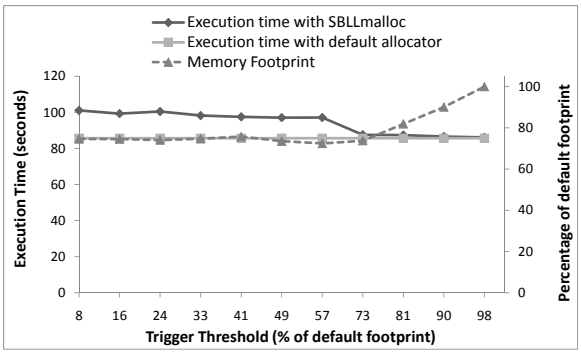


(b) Task scaling results

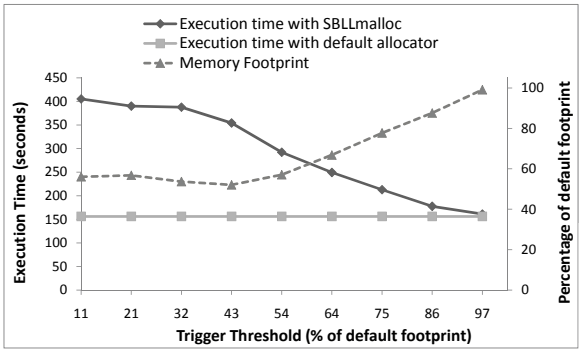
Fig. 17: IRS scaling results



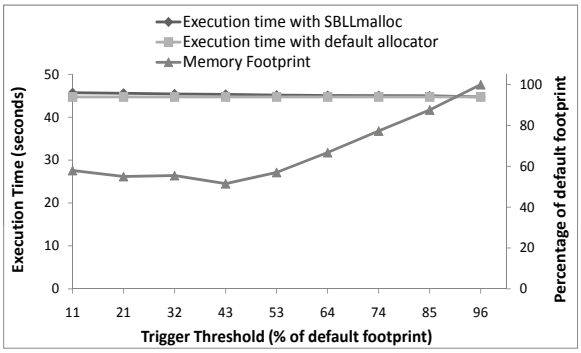
(a) IRS



(b) LAMMPS



(c) Chombo



(d) ParaDiS

Fig. 18: Execution time versus memory footprint reduction trade-offs

redundant data as much as increasing the number of tasks. The similarity in LAMMPS arises from zero pages associated with its communication routines. The computation and memory usage in LAMMPS scale with N/P where N is the number of molecules and P is the process count. However, communication scales as $(N/P)^{(2/3)}$. Thus, LAMMPS exhibits poorer problem size scaling than other applications.

We use only two ParaDiS input sets: *screw* and *form*. Figure 15 shows that *SBLLmalloc* reduces the memory footprint up to 53% and 72% for these data sets.

The AMR Godunov solver from the Chombo application achieves larger benefits as problem size increases (Figure 16(a)). Similarly, Figure 16(b) shows that increases in tasks per node also lead to more benefit. Specifically, we observe $40-44\%$ memory footprint reductions while running eight Chombo tasks per node. We also experimented with the AMR Elliptic solver from Chombo, where we observe around 50% reduction in memory footprint, which arose solely from merging zero pages. We are investigating the behavior with this solver further, so we do not include those graphs.

The problem size in IRS is increased by changing the number of zones per domain or the number of domains. Figure 17(a) shows that memory footprint reductions are nearly constant as we change the number of domains. Figure 17(b) indicates that increasing the task count per node gives larger benefit and *SBLLmalloc* can reduce the memory footprint by over 32% with a task per core per node. We experiment with different merge thresholds for IRS as well (Figure 18(a)), and

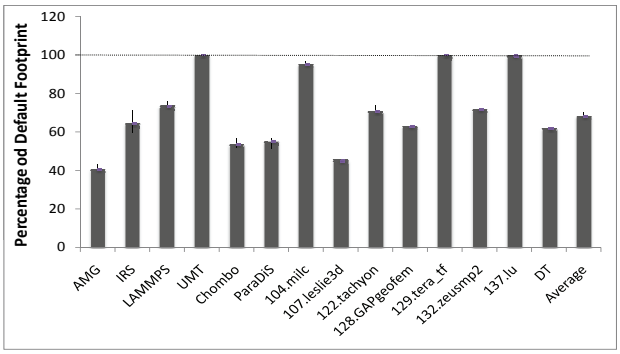


Fig. 19: Reduction results for all benchmarks

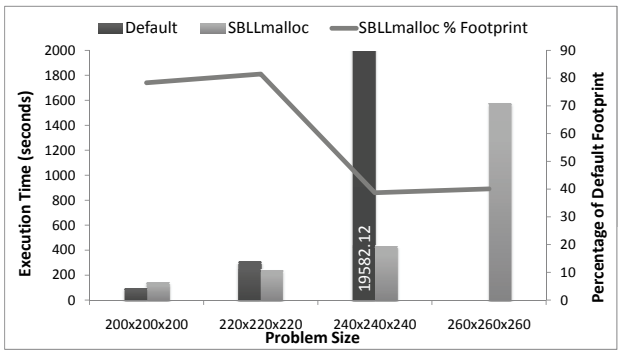


Fig. 21: AMG Swap Overhead

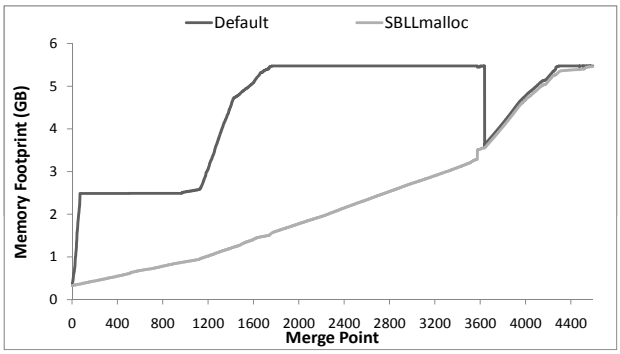


Fig. 20: Memory usage over UMT lifetime

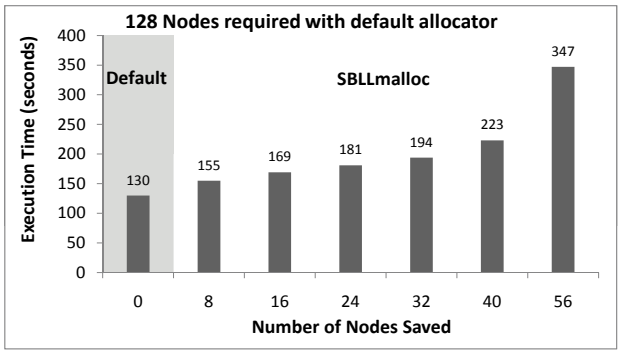


Fig. 22: AMG large run trade-off

we observe that we can reduce the memory footprint by 40% in exchange for higher overhead (33.33%). Using a better trade-off point, we can still reduce the memory footprint by 23.3% with an execution time overhead of 12.89%. We present the trade-off results for other benchmarks in Figures 18(b), 18(c) and 18(d). Suitable trade-off points provide significant memory footprint reductions for these applications. Again, *SBLLmalloc* targets cases in which traditional memory management prevents running problem sizes due to available physical memory constraints. Therefore, as shown in Figure 13(d), *SBLLmalloc* does not degrade performance when the memory requirement is low, but enables running large problems using the same amount of physical memory.

SPEC MPI2007 executes the benchmarks with multiple inputs, so we present the results as an average with error-bars, as shown in Figure 19. We observe that most of these benchmarks show significant data similarity being exploited by *SBLLmalloc*. However, *SBLLmalloc* provides little benefit to UMT (only 0.3% as Figure 20 shows). Similar behavior is observed in *129.tera_tf* and *137.lu*. Thus, we observe that some applications do not exhibit significant similarity that *SBLLmalloc* can exploti. However, applications that use large matrices typically benefit from *SBLLmalloc*. For the 14 benchmarks that we studied, *SBLLmalloc* reduces memory footprint by over 32% on average and up to 62%.

B. Beyond the Limit of Memory Capacity

We have shown the potential to run problems larger than the actual capacity of a node with our experimental results in Section V-A. In this section we compare *SBLLmalloc* to using disk space to increase attainable problem sizes. We then perform experiments that demonstrate how *SBLLmalloc* effectively increases the system size for IRS and AMG.

Swapping, which could serve as an alternative to *SBLLmalloc*, has high overhead due to the high latency of disk accesses. As the compute nodes in our experimental HPC system lack hard disks, we cannot perform swap-space experiments on it. Therefore, we performed an experiment on a 64-bit RedHat OS based Linux server with $6GB$ of physical memory. We ran large AMG problems, starting from problems that have low memory requirements to ones that require more than the available physical memory. Figure 21 shows that when the memory footprint is more than 80% ($4.8GB$) but less than $6GB$, *SBLLmalloc* incurs runtime overhead while a swapping solution does not since the problem still fits in the available memory. However, we observe more than $45X$ overhead with swapping for problems that require more than the available physical memory. An AMG problem of size $260 \times 260 \times 260$ did not complete in 2 days whereas *SBLLmalloc* enabled it to complete within less than 30 minutes.

IRS requires a cubic number of cores. Therefore, we experiment with 512 cores in a 64-node subset, which can fit at most a problem with $884,736,000$ zones. With *SBLLmalloc*, we can run a problem with $1,073,741,824$ zones, i.e., a problem 21.36% larger while using the same hardware resources.

We exhaust the entire memory of 128 nodes by fitting an AMG problem size of $2096 \times 1048 \times 1048$. An attempt to run even larger problems without using *SBLLmalloc* results in an

```
// matrix diagonal data matrix allocation
// malloc 584278016 0x2aaac2ecf000
diag_data =
      hypre_CTAlloc(double, diag_i[local_num_rows]);

// matrix diagonal data index allocation
// malloc 292139008 0x2aaab1834000
diag_j = hypre_CTAlloc(int, diag_i[local_num_rows]);
```

Fig. 23: AMG large matrix data and row index allocation

out-of memory error. With *SBLLmalloc* the same size problem can be solved using just 72 nodes, requiring 43.75% fewer nodes. We show the trade-off between execution speed and node usage in Figure 22.

C. Characterization of Similarity

Perhaps counterintuitively, we find significant *dynamic* data similarity across MPI processes in a wide set of applications. Gaining insight to the data and code structures that exhibit this similarity not only could reshape our intuition but also identify typical algorithmic patterns that *SBLLmalloc* particularly suits. Thus, we might identify new applications that would benefit from our approach. For this reason, we analyze three representative applications: AMG, which benefits most from *SBLLmalloc*; LAMMPS, which shows moderate similarity from primarily zero pages; and 122.tachyon, a parallel ray-tracing application, which exhibits moderate data similarity.

AMG uses the Algebraic Multi-Grid kernel to solve partial differential equations (PDEs). In this process it solves smaller, related problems to reach an approximate solution, which reduces the iterations required to solve the target problem. These matrices are in a sparse matrix format in which the actual data and the row indices are stored in vectors. As shown in Figure 23, these two data structures are very similar across MPI tasks, where the largest allocation corresponds to the actual data matrix and the second largest one corresponds to the row indices. Similarly many other allocations for initializing large matrices are dynamically merged and written as Figure 24 shows. Our knowledge of the application indicates that that this similarity arises from inherent structure in the matrices that correspond to the PDEs.

Figure 25 shows the merging and writing profiles of three representative allocations in a single MPI task. The largest one, which the bottom profile shows, corresponds to Figure 23. In this figure, the Y coordinate shows the page addresses of each allocation. A *blue* point in the figure indicates merging of a page and a *black* point at the same Y coordinate further along the X axis, which represents time, indicates a write to that page that causes its unmerging. The first set of blue points in a vertical line corresponds to merging after `calloc` initializes the pages to zeros. A tilted black line indicates the initialization of this region, after which the merging process starts again, merging the matrix for almost the entire execution. The other two profiles correspond to allocations described in Figure 24. Importantly, many pages incur multiple merge and unmerge events, which demonstrates that a technique such as *SBLLmalloc* is required to exploit the similarity.

```
// strenth matrix generation
// malloc 292139008 0x2aaaec445000
hypre_CSRMatrixJ(S_diag) =
      hypre_CTAlloc(int, num_nonzeros_diag);

// Ruge Coarsening routine
// malloc 281165824 0x2aab007e4000
ST_j = hypre_CTAlloc(int,jS);

// interpolation process matrix data allocation
// malloc 73154560 0x2aab0628f000
P_diag_data = hypre_CTAlloc(double, P_diag_size);

// new matrix allocation
// in hypre_BoomerAMGInterpTruncation
// malloc 73064448 0x2aab0d6e3000
P_diag_data_new = hypre_CTAlloc(double,P_diag_size);

// allocation for matrix transpose operation
// in hypre_CSRMatrixTranspose
// malloc 73064448 0x2aaaeddf5000
AT_data = hypre_CTAlloc(double, num_nonzerosAT);

// hypre_BoomerAMGBuildCoarseOperator:
// RAP data allocation
// malloc 72818688 0x2aaaf3a6a000
RAP_diag_data = hypre_CTAlloc(double, RAP_diag_size);
```

Fig. 24: Vectors allocated to solve smaller problems

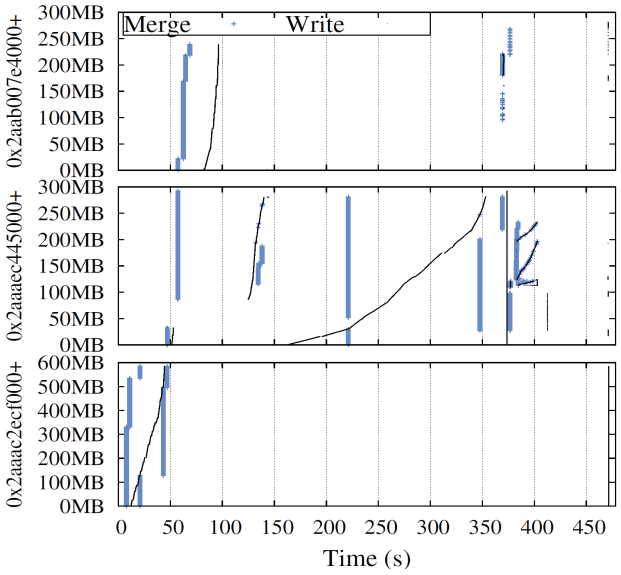


Fig. 25: Representative AMG merge and write profiles

LAMMPS is a classical molecular dynamics simulation. At the end of each time step, every MPI process makes a list of nearby atoms, which it transmits to neighboring processors. These messages are then unpacked, as Figure 26 shows. Merge and unmerge profiles indicate that these unpacked data pages exhibit similarity across MPI tasks. They primarily are zero-pages although a small fraction (≈10%) are non-zero pages.

122.tachyon is an image rendering application from SPEC MPI 2007 that shows moderate data similarity. Interestingly, this similarity is primarily from sharing non-zero pages. We show the merge and write behavior of large regions for this application in Figure 27, and the explanatory code snippet in Figure 28. While rendering an image, all MPI tasks allocate

```
// malloc 43847680 0x2aaacad3f000
special = atom->special =
          memory->grow_2d_int_array(
            atom->special,nmax,
            atom->maxspecial, "atom:special");

// malloc 30355456 0x2aaaccaa4000
dihedral_type = atom->dihedral_type =
          memory->grow_2d_int_array(
            atom->dihedral_type,nmax,
            atom->dihedral_per_atom,
            "atom:dihedral_type");
```

Fig. 26: Contributors to LAMMPS similarity

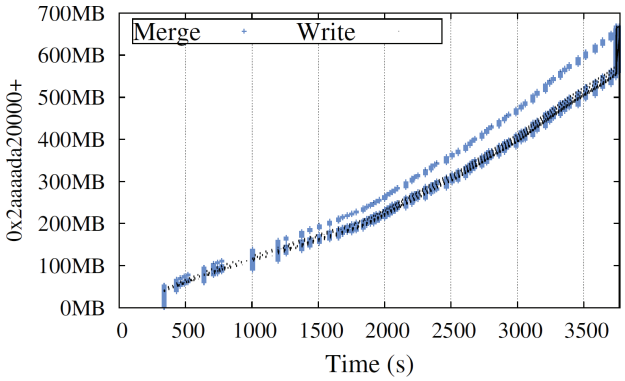


Fig. 27: Key 122.tachyon merge and write profiles

a buffer to hold a part of the image. This data structure exhibits significant similarity across tasks. We performed this experiment with multiple data sets to observe the same behavior. In function `rendercheck()` all MPI tasks of 122.tachyon allocate large buffers to hold the rendered scene. As the execution progresses, this buffer is modified and then merged as illustrated by the sequences of blue, black and blue points that correspond to the same page location in the middle profile in Figure 27.

These three applications provide useful insight into data similarity across MPI processes. IRS, which is dominated by a conjugate gradient solver, has a similar underlying algorithmic nature to AMG so we expect to observe the significant data similarity that it exhibits. Like LAMMPS, many other applications benefit from zero pages. For example, the AMR Elliptic solver obtains around 50% memory reductions due to sharing of zero pages. Applications like 122.tachyon that have very regular access patterns often have high, dynamic data similarity across MPI processes. We have also examined a parallel version of another popular ray tracing application, POV-Ray. We observe significant similarity even in small runs. However, we do not present this application due to inavailability of large input data sets, although it incurs a large memory footprint to render complicated scenes. We will characterize the data similarity in more applications as part of our future work.

D. Summary

Our study shows that *SBLLmalloc* reduces the memory footprint of running MPI tasks on multicore nodes by exploiting their inherent data similarity. Also, several applications

```
// malloc 669536256 0x2aaaada20000
scene->img = (unsigned char *) malloc(
        scene->hres * scene->vres * 3);
```

Fig. 28: Contributors to 122.tachyon similarity

show high data similarity from zero pages. For example, the Chombo AMR Elliptic solver demonstrates about 50% reduction in memory footprint only from zero pages while running 8 MPI tasks per node. *SBLLmalloc* reduces the memory footprint across all applications with which we experimented by 32% on average with each node running eight MPI tasks. *SBLLmalloc* enables problem sizes that are larger than can be run with the default memory allocation method using the same node count or would require many more nodes.

VI. Related Work

SBLLmalloc builds upon prior research in multiple domains. We adapt the mechanisms used in user-level distributed shared memories and perform content-based page merging. Our preliminary work [6] established the basic ideas behind *SBLLmalloc* while this paper contains much greater detail about the implementation and its optimization and presents a thorough evaluation.

DSM systems [9, 14, 15, 16, 18, 20] use similar virtual memory management techniques with a different goal. We optimize memory usage within a node. DSM systems provide the shared memory programming paradigm across nodes by sharing some physical memory. Thus, nodes can access a large pool of shared memory along with limited in-node memory. User level libraries implement software DSMs, similarly to *SBLLmalloc*. However, DSMs provide limited benefits to HPC systems due to latency and coherence traffic across nodes. *SBLLmalloc* can provide significant HPC benefits by effectively increasing total system memory size by implicitly sharing data pages within a node and, thus, avoiding the need for complex software coherence protocols in contrast to DSM's explicit data sharing across nodes.

Several researchers have investigated in-memory compression techniques, which are complementary to *SBLLmalloc*, in order to increase the effective DRAM capacity. Operating systems such as Sprite [10] used compression to reduce memory usage. Wilson *et al.* proposed efficient hardware structures [24] for faster compression and decompression to reduce the overheads. A compression cache [22] for Linux was proposed by Tuduce *et al.* for managing the amount of memory allocated to compressed pages. Extending *SBLLmalloc* with this technique on infrequently used pages, which we keep as future work, could reduce memory footprints even further.

Most OSs support some *copy-on-write* mechanism, which simply and efficiently limits identical read-only data to a single copy. However, simple *copy-on-write* techniques cannot support merging of duplicate but modified data. We exploit these techniques to detect when identical pages diverge. Waldspurger [23] incorporated searching for identical data in virtual machines (VMs) such as VMWare's ESX server,

Xen VMM [17]. Their technique performs content-based page searching by creating hashes from page contents and using hash collisions to detect identical pages. Disco [7] incorporated *transparent page sharing* in the OS by explicitly tracking page changes in kernel space. Our system, which does not require any OS modifications, performs byte-by-byte comparisons similarly to the ESX server.

Biswas *et al.* [5] studied the similarity across multiple executions of the same application, as several scenarios exhibit including machine learning and simulations. *SBLLmalloc* could apply to them. "Difference Engine" (DE) uses collisions of hashes computed from the page contents to identify identical pages under Xen VMs [13]. DE reduces memory footprints significantly when applied to infrequently accessed pages and combined with partial page merging. However, these approaches require special hardware or OS components while *SBLLmalloc* only requires a user level library.

VII. Conclusions

We have presented *SBLLmalloc*, a user-level library that efficiently manages duplicated memory across MPI tasks on a single node, for use in applications in which the problem size is limited by main memory size (due to cost or reliability concerns). *SBLLmalloc* uses system calls and interrupts to perform its work, requiring no OS or application changes. We perform two optimizations: identifying zero pages across and within tasks and finding identical pages at the same virtual address across tasks. Identifying identical data and merging pages incurs high overhead. With optimizations and careful choices on when to merge, we achieve near native execution speed with substantial memory footprint reduction.

On average we observe a 32% reduction in memory footprints. *SBLLmalloc* supports problem sizes 21.36% larger for IRS than the standard memory allocator while using the same resources. Alternatively, we can reduce the nodes required to run a large AMG problem by 43.75%. Both zero pages and identical pages contribute to the overall performance. Our analysis demonstrates that the the data similarity across MPI processes is dynamic and the sources vary significantly across application domains. Nonetheless, most domains exhibit some dynamic similarity, which *SBLLmalloc* provides the mechanism to exploit for memory footprint reductions.

Acknowledgements

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-468331), and the National Science Foundation under CAREER Grant No. 0855889, CCF-1017578 and MRI-0619911 to Diana Franklin, Grant No. FA9550-07-1-0532 (AFOSR MURI), a Google Directed Research Grant for Energy-Proportional Computing and NSF 0627749 to Frederic T. Chong, Grant No. CCF-0448654, CNS-0524771, CCF-0702798 to Timothy Sherwood.

References

[1] SPEC MPI2007: http://www.spec.org/mpi2007/.

[2] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/.

[3] Applied Numerical Algorithms Group. Chombo Software Package for AMR Applications Design Document. https://seesar.lbl.gov/ANAG/chombo/ChomboDesign-3.0.pdf, April 2009.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, V. Venkatakrishnan, S. Weeratunga, S. Fineburg, and H. Simon. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[5] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-Execution: Multicore Caching for Data-Similar Executions. In *Proceedings of the International Symposium on Computer Architectures (ISCA'09)*, June 2009.

[6] S. Biswas, D. Franklin, T. Sherwood, F. T. Chong, B. R. de Supinski, and M. Schulz. PSMalloc: Content Based Memory Management for MPI Applications. In *MEDEA '09: Proceedings of the 10th International Workshop on MEmory performance: DEaling with Applications, systems and architecture*, 2009.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[8] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 19, Washington, DC, USA, 2004. IEEE Computer Society.

[9] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *In Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[10] F. Doughs. The Compression Cache: Using On-Line Compression To Extend Physical Memory. In *Proceedings of the USENIX Winter Technical Conference*, pages 519–529, 1993.

[11] J. E. Garlick and C. M. Dunlap. Building CHAOS: an Operating Environment for Livermore Linux Clusters, February 2002. UCRL-ID-151968.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[13] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008)*. USENIX, December 2008.

[14] W. Hu, W. Shi, Z. Tang, Z. Zhou, and M. Eskicioglu. JIAJIA: An SVM System Based on a New Cache Coherence Protocol. Technical Report TR-980001, Center of High Performance Computing , Institute of Computing Technology, Chinese Academy of Sciences, 1998.

[15] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *ACM SIGARCH Computer Architecture News*, 20(2):13–21, 1992.

[16] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–131, 1994.

[17] J. Kloster, J. Kristensen, and A. Mejlholm. On the Feasibility of Memory Sharing: Content-Based Page Sharing in the Xen Virtual Machine Monitor. Master's thesis, Department of Computer Science, Aalborg University, June 2006.

[18] K. Li. Ivy: A Shared Virtual Memory System for Parallel Computing. *Distributed Shared Memory: Concepts and Systems*, page 121, 1997.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[20] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, pages 95–106, 1997.

[21] R. Stevens, A. White, P. Beckman, R. Bair, J. Hack, J. Nichols, A. Geist, H. Simon, K. Yelick, J. Shalf, S. Ashby, M. Khaleel, M. McCoy, M. Seager, B. Gorda, J. Morrison, C. Wampler, J. Peery, S. Dosanjh, J. Ang, J. Davenport, T. Schlagel, F. Johnson, and P. Messina. A Decadal DOE Plan for Providing Exascale Applications and Technologies for DOE Mission Needs, 2010. https://asc.llnl.gov/ascpihome/post-meeting.html.

[22] I. Tuduce and T. Gross. Adaptive Main Memory Compression. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[23] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[24] P. Wilson, S. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 101–116, 1999.