Memory Tagging in Charm++

Filippo Gioachin
Department of Computer Science
University of Illinois at Urbana-Champaign
gioachin@uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu

ABSTRACT

Many scientific applications are logically decomposed into *modules*, each module performing a different type of computation. These modules are then linked together inside the same executable. While these modules are logically independent, they are not physically independent: a faulty module can corrupt the state of another one.

By identifying the different modules inside an application, tagging the memory according to the different modules, and performing extra runtime checks, we can automatically detect certain type of errors. We implemented our idea inside the CHARM++ runtime system, where modules can be easily identified. We illustrate the validity of our approach, and evaluate its overhead.

Keywords

Debugging, Parallel Debugging, Memory Debugging, Memory Tagging

1. INTRODUCTION

When an application is decomposed into independent modules that share the same resources, the possibility of resource conflicts arise. Memory is a common example of a shared resource. In a correct decomposition, each module stores the state necessary to perform its task in memory, and uses some specific area to exchange information with other modules. Since the virtual address space is common, all the modules have access to the entire address space. Therefore, they can modify the state of other modules. While this cannot be prevented, it breaks the abstraction that modules are independent, and makes faulty modules difficult to identify. The definition of module depends on the programing model used. It can be chares, as we shall see later for CHARM++ applications, object files, or libraries.

Consider a parallel program for the simulation of galaxy formation. This program will likely have multiple separate modules to compute the various forces present in the universe: gravitational, hydrodynamic (SPH), magnetodynamic, etc. For good performance, these modules may run "simultaneously", thus allowing the communication of one module to be overlapped with useful computation of another module. By simultaneously we mean that any given processor participates in the computation of all the different forces, interleaving their execution over time. All these modules will need to update the memory storing the final forces acting on the simulated portion of space. This memory is therefore used to exchange information. In addition, each module will also have its private data to be used during the computation phase. This private data should be modified exclusively by the owner module. If, for example, the gravity module accesses and modifies the data stored for the SPH computation, we want to notify the user that the gravity module is misbehaving, and might be faulty.

By creating a tagging system where each allocated memory block is marked with an identifier of the module that uses it, it becomes possible to intercept modifications of such memory by other modules. The user can be notified of these misuses and can determine if they are valid, such as in the case of the final forces in the previous example, or if they are not, and therefore identify the faulty module.

This technique is general for both sequential and parallel applications that use independent modules to integrate different codes within the same application. In this paper, we describe an implementation of this technique within the CHARM++ runtime system. CHARM++ is described in detail in Section 2. In Section 3, we describe the CHARMDEBUG debugger, and in Sections 3.2 and 3.3, we show how we can use this combined infrastructure to detect certain common problems. Later, in Section 4, we analyze the overhead we introduced, and in Section 5, we compare our technique with other works in the field. We conclude with final remarks and future work.

2. THE CHARM++ RUNTIME SYSTEM

CHARM++[7] is a popular runtime system for developing parallel applications. Several applications have been developed based on the CHARM++ runtime system. Among these are NAMD[1], a molecular dynamics code and winner of the Gordon Bell award in 2002, OpenAtom[2] for Car-Parrinello ab-initio molecular dynamics and ChaNGa[4] for cosmological simulations. The combined workload of these application accounts for more than 15% of the time spent executing jobs

PADTAD - Workshop on Parallel and Distributed Systems: Testing and Debugging, July 20-21,2008

on several NSF funded supercomputers in the United States.

The primary concept of CHARM++ is object virtualization[6]. In CHARM++, the user divides the computation into small objects, called *chares*. These chares are assigned to the available processors by the runtime system itself, thus allowing load balancing[14] and other automatic performance optimizations such as communication optimization[8].

These chares communicate with each other via asynchronous messages. Messages trigger function calls on the destination chare. These functions are called *entry methods*. The computation performed by an entry method upon receipt of a message depends on the information carried by the message and the internal state of the chare receiving that message. Chares performing similar operations can be grouped into collections of chares. These collections are able to be indexed, and are also referred to as *chare arrays*. While each chare is an independent entity, all chares in a collection share the same *chare type*. All chares are considered independent modules, regardless of their type. This is so because their state is independent.

In the scenario of the cosmological simulation described earlier, the program will consist of several different collections of chares, one for each force computation performed during the simulation, such as SPH, gravity, etc. Each collection will consist of many chares, possibly thousands or even millions, and each chare will perform a specific force computation on a small portion of the simulation space.

2.1 The Memory Subsystem

In CHARM++, the memory subsystem, i.e the implementation of the malloc, free and other memory related functions, is included in a shared library. This allows CHARM++ to implement multiple versions of the memory library, and enables the user to choose which one to use at link-time (see Figure 1). The default version, gnu.o in the Figure, does not have any debugging support and is meant for production usage. This version is based on the glibc memory allocator. Another memory library, os.o, does not implement the memory functions, and lets the user link to the default one provided by the system. All the others are based on the glibc standard library, as the default implementation, but they re-implement the memory functions (malloc, free, realloc, etc.) and use the glibc ones internally.

To allow multiple memory libraries with different capabilities to be based on the same underlying glibc memory allocator while avoiding the problem of re-implementing the entire allocator, the glibc routines have been renamed, prepending them with an "mm_" prefix. Any memory library wrapping around glibc allocator will define the functions malloc, free, etc, and internally use the "mm_" versions. For example, the default malloc simply calls mm_malloc, while another malloc implementation can decide to allocate extra size for internal usage of the library itself, or to fill the newly allocated memory with certain patterns.

Some memory implementations inside CHARM++ are named in Figure 1. *Paranoid* provides buffer overflow detection by allocating extra space at both sides of the user allocation and filling it with a predefined pattern. At deallocation, these

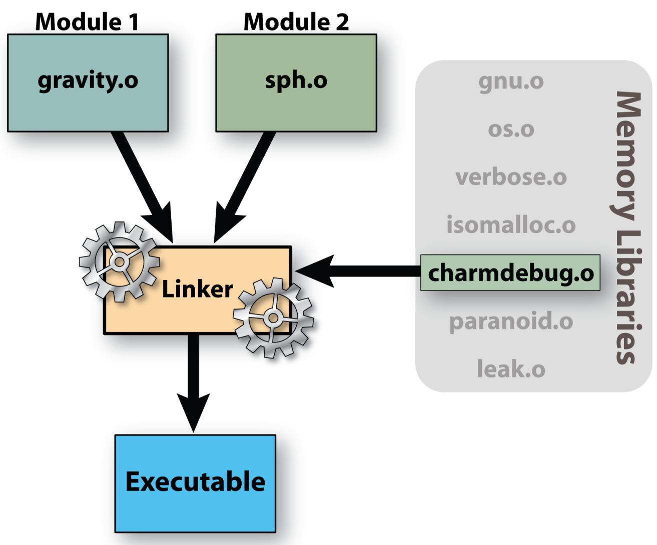


Figure 1: Application linking stage. The application modules are linked together with one Charm++ memory implementation to produce the executable.

extra spaces are checked for modifications. Moreover, deallocated regions are also filled to help detect usage of dangling pointers. *Leak* allows the user to mark all allocated blocks as "clean", and later performs a scan to see if new memory blocks were allocated. This is useful in iterative programs, where the total memory over various iterations should not increase. This assumes that the code does not reallocate new memory at every iteration. The CHARMDEBUG memory module is used in conjunction with the CHARMDEBUG debugger, and is described in more detail in Section 3.1.

3. CHARMDEBUG

CHARMDEBUG[5] is a graphical tool written in Java that allows the user to remotely debug parallel applications written in CHARM++. In a typical scenario, the user will start CHARMDEBUG on his own workstation. Then, through CHARMDEBUG, the user can select and start an application on a remote parallel cluster where it has been previously compiled. Inside CHARM++ there is a built-in CHARMDEBUG plugin. The combination of the graphical tool and this plugin allows the user to visualize information pertinent to its code. Such information includes, but is not limited to, the messages queued in the system, the chares present on a processor, and the state of any chare. Moreover, the user can set breakpoints at the beginning of entry methods. While CHARM++ can be run on top of MPI, tools specific to MPI applications, such as TotalView[13], will unhelpfully provide the user with more information regarding the CHARM++ internal implementation, than regarding the user's code.

CHARMDEBUG and the CHARMDEBUG plugin communicate through a high-level client-server protocol, called Converse Client-Server (CCS). CCS is a standard for CHARM++ and is built into all CHARM++ applications. CCS is a stateless connection where a client can send requests to the running application, and the application will receive the request as a message to a pre-specified entry method. The application can perform any action as a consequence of such a message, and finally return an answer to the client if it so chooses. The CharmDebug plugin is responsible for managing the incoming requests from CharmDebug, and replying to them with the appropriate information.

Since the communication between the debugger and the application under examination happens through a single high-level connection, no connection is necessary to each individual process of the parallel application. This allows CharmDebug to scale to as large a configuration as Charm++ does. Even if a direct connection is not established to each processor, the user can request the debugger to open a GDB session for a particular processor. This lets the user descend to a lower level and perform operations that are currently not directly supported by CharmDebug, such as stepping through the source code and inspecting values at points inside an entry method.

3.1 The CharmDebug Memory Library

Among the memory libraries described in Section 2.1, we mentioned one built specifically for use with CharmDebug. This library, for every memory block requested by the user, allocates some additional space for internal usage. The details of the extra space allocated are shown in Figure 2, and are described throughout this section. The layout refers to 64-bit machines. The library returns to the user a pointer to the white region marked *user data* in the Figure.

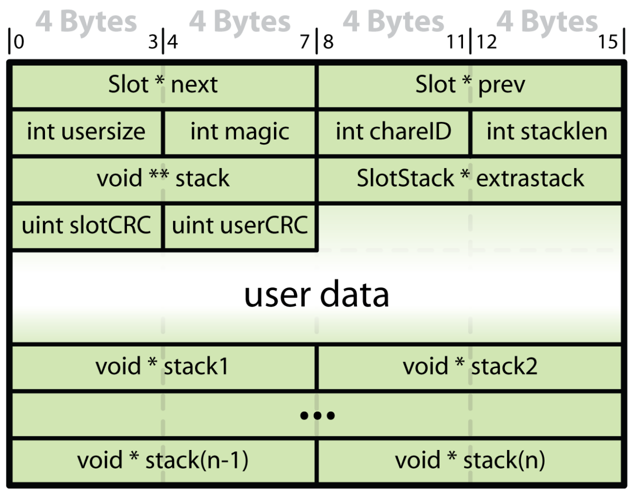


Figure 2: Layout of the extra space allocated by the CharmDebug memory library on 64 bit machines. The memory allocated for CharmDebug purposes is shown in shaded color, the memory for the user is shown in white.

We built upon the CharmDebug existing framework to provide support for debugging memory-related problems. The CharmDebug memory library extends the existing CCS requests that the CharmDebug plugin can serve by providing extra information regarding the memory status. A simple operation that CharmDebug can request is to view the memory of any given processor. Figure 3 shows how CharmDebug visualizes the information received from the application through CCS. The application in the Figure performs a simple Jacobi computation on a two-dimensional matrix. Each allocation is colored in one of four different colors, according to its usage:

- memory that is occupied by a specific chare (in yellow);
- a message sent from one chare to another (in pink);
- memory allocated on the heap by the user code (in blue);
- memory allocated for the use of the Charm++ runtime system (in red).

Moreover, the CharmDebug memory library automatically collects stack trace information at every point where the user requested memory allocation. This information is stored at the end of the user buffer, as shown in Figure 2. The user can see this information at the bottom of the memory allocation view (Figure 3) by moving the mouse pointer over the allocated blocks.

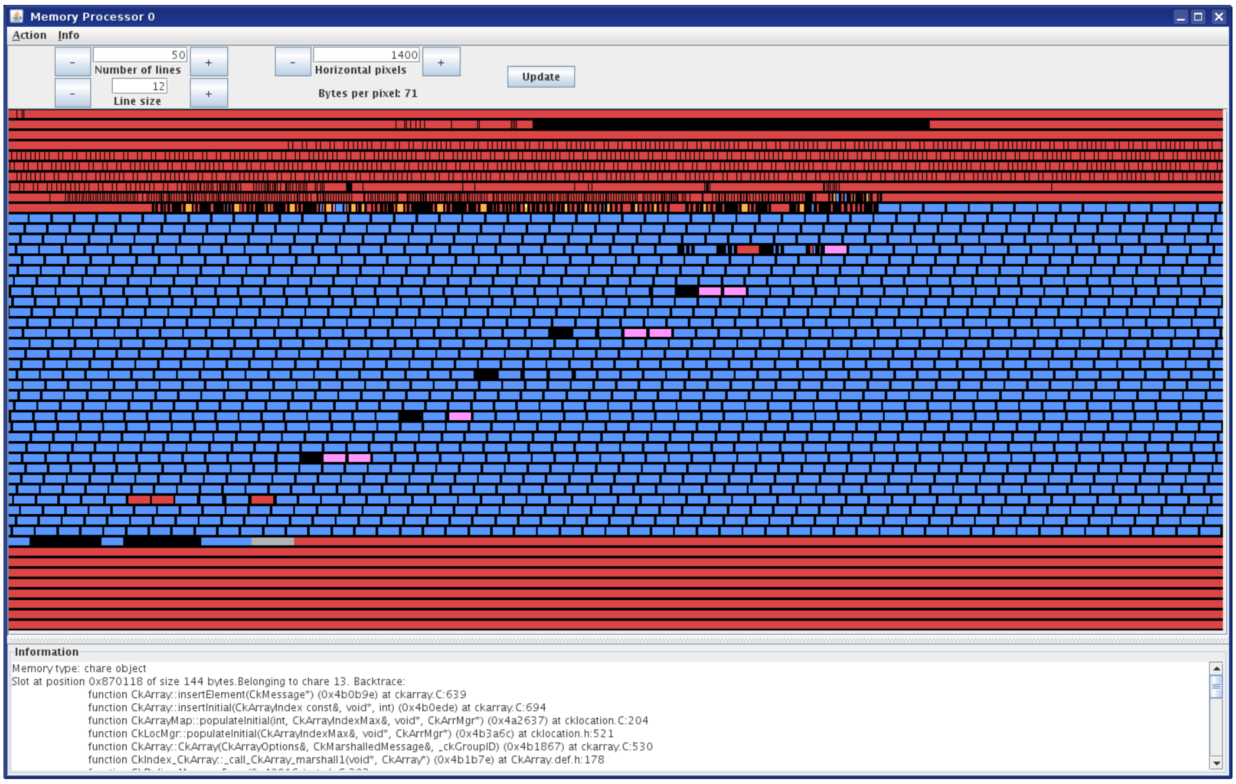
Stack traces can also be combined by CharmDebug into *allocation trees*. An allocation tree is a tree rooted at the routine starting the program, typically main or a loader library routine. The children of a node are the functions that were called by the function represented by that node. The leaves are functions which called the malloc routine. This tree can become a forest if not all stack traces start from the same routine. This can happen, for example, in the presence of user-level threads with independent stacks. CharmDebug can construct an allocation tree for a single processor or for a subset of them. Allocation trees can be used for statistical analysis to provide insight of memory problems.

One of the operations that the CharmDebug memory library can perform is memory leak detection. Each processor parses stack and global variable locations for pointers to heap data. The heap memory blocks reachable by those pointers are further parsed for more pointers. This continues until all reachable locations are detected. Blocks not reachable are declared leaks. The result is reported in the same memory view described earlier (not shown here).

In the Charm++ environment, the user code always runs in the context of some chare. These chares, as we have seen in Section 2, are independent of each other, and should not interact except through messages. Therefore, another tag is automatically associated to each memory block, to identify which chare allocated it. This tag is shown in Figure 2 as *chareID*. Figure 4 shows the same Jacobi program with the highlighting of the memory associated with a particular chare. We will see in the following subsections how this tagging can be used to identify certain type of bugs.

3.2 Detecting cross-object memory modifications

In a program like Jacobi, suppose chare A allocates a memory block for its local matrix, and then passes a pointer to the last row to chare B, instead of a newly allocated message with a copy of the last row inside. Chare B can access and modify the matrix of chare A during its computation. Nevertheless, Chare B is not supposed to modify chare A's state.



```
Memory type: chare object
Slot at position 0x870118 of size 144 bytes.Belonging to chare 13. Backtrace:
                    function CkArray::insertElement(CkMessage*) (0x4b0b9e) at ckarray.C:639
                    function CkArray::insertInitial(CkArrayIndex const&, void*, int) (0x4b0ede) at ckarray.C:694
                    function CkArrayMap::populateInitial(int, CkArrayIndexMax&, void*, CkArrMgr*) (0x4a2637) at cklocation.C:204
                    function CkLocMgr::populateInitial(CkArrayIndexMax&, void*, CkArrMgr*) (0x4b3a6c) at cklocation.h:521
                    function CkArray::CkArray(CkArrayOptions&, CkMarshalledMessage&, _ckGroupID) (0x4b1867) at ckarray.C:530
                    function CkIndex_CkArray::_call_CkArray_marshall1(void*, CkArray*) (0x4b1b7e) at CkArray.def.h:178
```

Figure 3: Memory view of the allocated memory for a simple hello world program. The color correspondence is: yellow-chare, pink-message, blue-user, red-system.

More generally, different chares are not supposed to modify each other's memory. Based on this concept, we define each memory allocated by a chare to belong to that chare, and only that chare will be allowed to modify its content. The only exception is messages whose ownership will be passed from the creator to the chare that the message is delivered to.

We use a cyclic redundancy check (CRC-32-IEEE) to detect when memory was modified by a chare different from its owner. Let us consider the example above. Suppose we compute the CRC for all allocated memory blocks and store them before we deliver to chare B the message containing the pointer passed by A. Subsequently, we deliver the message to B and let it perform its computation. Let us assume B immediately uses the pointer to modify the matrix of A (if B uses the pointer in another entry method later, the same discussion applies for that entry method). After the entry method of B terminates, we recompute all CRCs. The CRC for the block containing A's matrix will not match as B modified it. Since the block belonged to A and not B, we raise an exception and notify the user. The CRC of some blocks belonging to B might also not match, but we ignore these as B was allowed to modify that memory. Currently, we simply notify the user of the memory violation and continue running the program. For the future, we envision an interface where the user can inspect the program when a violation occurs, and then decide if it is safe to ignore it. All the CRCs are stored as part of the extra space allocated by the CharmDebug memory library, as depicted in Figure 2 by the field *userCRC*.

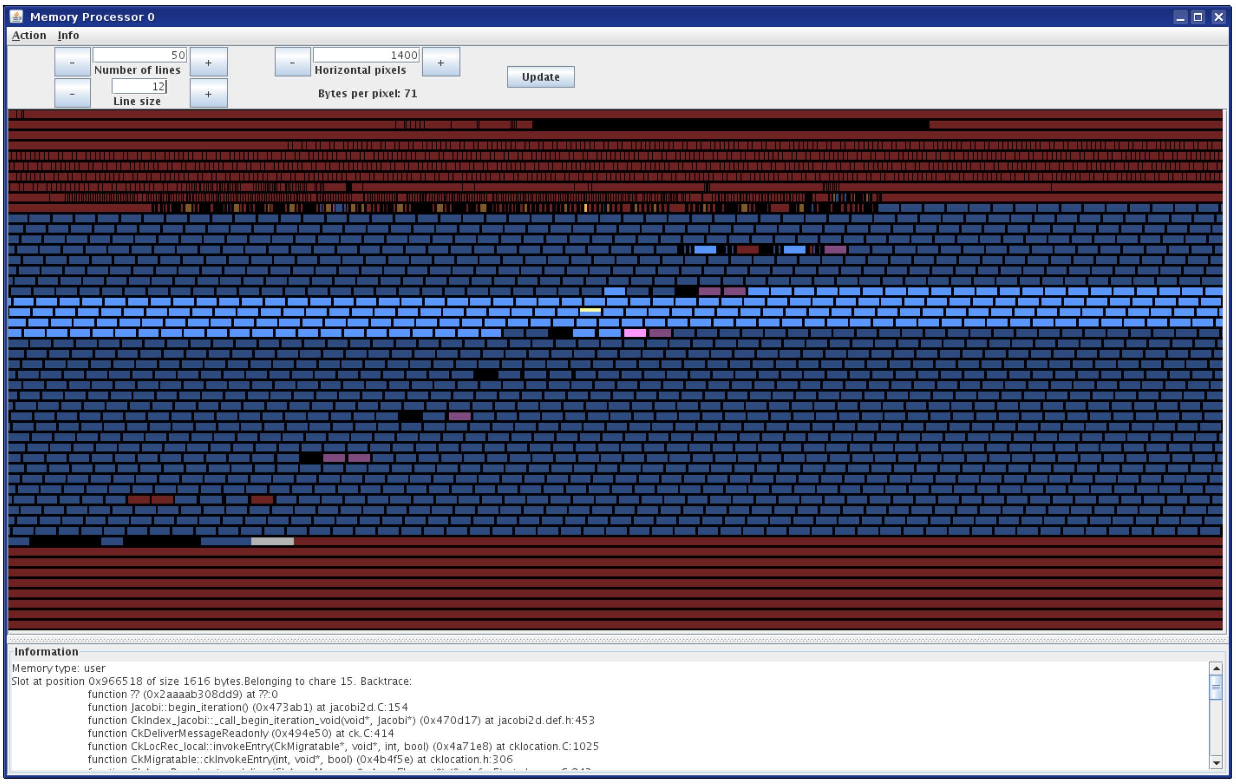
More generally, by computing all the CRCs before the user entry methods are called, and rechecking them after the user entry methods return, we can restrict the portion of code that is faulty to a specific entry method. The user is then provided with useful information about the error: which chare and which entry method are responsible for the modification, which memory block has been modified, and to whom that memory block belongs.

In situations where the user suspects certain chares or entry methods to be the cause of the fault, only those can be instrumented to perform this check, thereby reducing the overhead. On the other hand, if all the entry methods are to be checked, by combining the check after a message delivery with the recomputation before the next message is delivered, the overhead can also be reduced.

One limitation is if the faulty entry method internally spans a large amount of code. In this situation the portion of code that has to be inspected for faults is still large. By allowing the user to request extra checks to be performed even in the middle of an entry method, the user can split the faulty code into subregions, and be notified about the region causing the exception.

3.3 Detecting buffer overflow

Another common problem in applications is the corruption of adjacent blocks of memory due to overrun or underrun of



Number of lines: 50
Line size: 12
Horizontal pixels: 1400
Bytes per pixel: 71

Information

```
Memory type: user
Slot at position 0x966518 of size 1616 bytes.Belonging to chare 15. Backtrace:
                function ?? (0x2aaaab308dd9) at ??:0
                function Jacobi::begin_iteration() (0x473ab1) at jacobi2d.C:154
                function CkIndex_Jacobi::_call_begin_iteration_void(void*, Jacobi*) (0x470d17) at jacobi2d.def.h:453
                function CkDeliverMessageReadonly (0x494e50) at ck.C:414
                function CkLocRec_local::invokeEntry(CkMigratable*, void*, int, bool) (0x4a71e8) at cklocation.C:1025
                function CkMigratable::ckInvokeEntry(int, void*, bool) (0x4b4f5e) at cklocation.h:306
```

Figure 4: Dimmed memory view. In brighter colors are shown the regions of memory allocated by a specific chare, in darker colors all the others.

array accesses. A typical debugging technique is to allocate extra memory at the two ends of the user buffer, fill (or paint) it with a predefined pattern, and check if this has been overwritten by the program. One problem with this method is the granularity with which to perform the checks on the painted areas. Checking only when a block is deallocated is not enough, as it may never be deallocated or, even if deallocated, the region of code containing the error cannot be exactly pinpointed. Additional periodic checks on all the memory blocks may reveal the problem at an earlier stage, but it would still not identify exactly which lines of code were responsible for the corruption.

As in the case for cross-object modifications, we can use the entry method boundaries to perform the buffer overflow checks. Since CHARMDEBUG already allocates extra space on both sides of the user allocated data, we utilize this same portion of memory to check for buffer overflow corruptions. One necessary change in the detection scheme is that we now cannot paint that memory with a predefined pattern, since it contains valid information. The second CRC field in Figure 2, *slotCRC*, operates on the extra memory allocated by CHARMDEBUG. As before, we compute all CRC on the extra memory before the user entry methods are called, and recheck them after the user entry methods return. If a mismatch is found during the recheck, the fault is attributed to the entry last entry method that executed.

Again, the user may specify a coarser granularity of checks, in which case there will be a set of messages that could have caused the problem, or a finer granularity by adding extra checks inside his code. This method can fail if the modification is such that the CRC is still correct after the modification. The probability of this happening is very low.

4. PERFORMANCE

We analyzed the overhead imposed by our implementation. The test application was a parallel program implementing a two-dimensional Jacobi computation. The total matrix was divided among chares organized in a two-dimensional chare array, where each chare was assigned a square portion of the matrix. To update the matrix for the next iteration of the algorithm, each chare exchanged the borders of its portion of the matrix with its four adjacent neighbors. We performed our benchmark on a dual-socket Intel Xeon 1.86 GHz quad-core workstation.

We timed the execution of the routine performing the CRC computations and checks. We repeated this test by varying the problem size, and therefore the total amount of total memory allocated per processor. Figure 5 plots the results. We can see that the time linearly increases with the amount of memory allocated. This is to be expected as the check has to perform a computation proportional to the amount of memory allocated. By extracting the slope of the curve, we obtain that each check accounts for an overhead of 4.3 milliseconds per megabyte of memory allocated.

Given the amount of work performed on average inside entry methods, the overhead can be computed analytically. As-

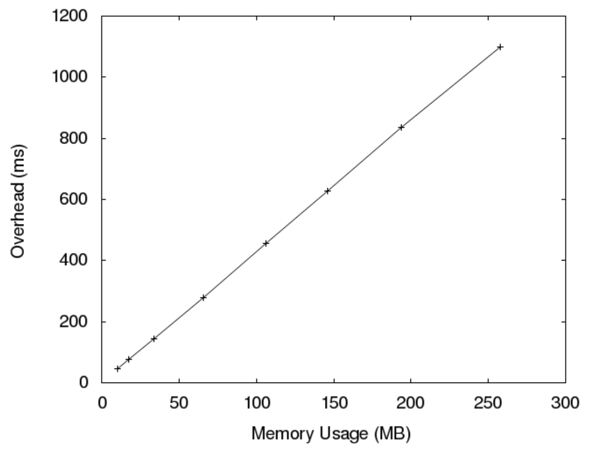


Figure 5: Overhead to compute the CRC for all the memory, at different problem sizes.

suming we perform the check for every entry method, if t is the entry method's average time in milliseconds, and M is the amount of memory allocated in megabytes, the execution time of the program with CRC checks is $1 + \frac{4.3 \cdot M}{t}$ times the time of the unchecked program. For example, with 50 MB of allocated memory, and an average entry method's execution time of 5 ms, the CRC checked execution time is 43 times the original one. This overhead is acceptable, but it quickly becomes too high for finer granularity applications. We are studying other techniques, such as leveraging the virtual memory paging system and its protection mechanisms, to allow us to reduce it.

In a parallel scenario, since CHARM++ is a distributed system, the memory in each processor is independent from that of other processors. Therefore, the overhead each processor incurs depends only on its allocated memory and the average computation time of the entry methods it executes. Overall, the total overhead in the parallel execution is the maximum overhead across all processors. To prove this, we ran several experiments on the same Jacobi application from one processor to eight processors. The results, not shown here, matched our expectation.

5. RELATED WORK

There are various tools that help debugging shared accesses to a variable in a multithreaded environment. Intel Thread Checker[3] is one such tool. It can detect both read and write unsynchronized accesses to shared variables. It uses dynamic instrumentation to inspect each memory access performed by each thread, and returns statistics on threads using the same locations. Given that it needs to intercept and perform extra operations at every memory access, it significantly slows down the execution of the program. An improvement on this tool has been proposed[10] by filtering most memory accesses that are not likely to produce data races, and check only those not filtered out.

Another tool is RecPlay[9] which combines record-replay and on-the-fly data race detection to efficiently inspect concurrent programs implemented using POSIX synchronization operations, and detect data races. The algorithm requires the program to be run several times to obtain all the information to identify both the racing data and the racing instructions. Even though the code is executed multiple times, since most of the time the program runs without slowdown, the total overhead is reduced. This algorithm has the disadvantage that it can only detect the first race condition, and loses effectiveness if the user decides that he does not want to, or cannot, remove the data race.

All these tools are for shared memory accesses. In the scenario described in this paper, where there is one single thread of execution, and the program is decomposable into independent modules, such tools would not be useful. Other tools for sequential programs, such as Valgrind[11], are capable of detecting buffer overflow and other memory-related problems. These tools typically incur an acceptable overhead. Again, for the scenario described here, these tools provide little support to the user.

TotalView[12] is another powerful debugging tool capable of inspecting and analyzing the memory allocated by an application, and it supports parallel distributed systems such as MPI. TotalView allows the user to collect memory views and save them for future reference. These saved views can be compared against each other or against the status of the live application. By saving and comparing memory states, the user can simulate our comparison tool. Nevertheless, it is not possible to automate the collection of states and their comparison, forcing the user to undertake a tedious process of stepping through the code. In situations where the error appears only after a significant amount of time, this approach becomes impractical. TotalView, by saving the complete memory state, can provide a more detailed difference report than our tool based on CRC codes. Nevertheless, saving the state requires high memory usage and disk bandwidth, slowing down the process.

6. SUMMARY AND FUTURE WORK

In this paper we presented a technique based on memory tagging to help detect memory problems in applications that use a module-oriented programming model. We applied this technique to applications written in CHARM++ and showed how this technique can be used to detect two types of memory problems: (1) when a module (chares in CHARM++) modifies memory belonging to other modules, and (2) when a buffer overflow occurs.

The current overhead that our implementation suffers is too high for most real situations. We are studying refinements to our scheme to improve the performance and significantly reduce the constant overhead time we described in the performance section. In particular, we are looking at two techniques. One technique is to duplicate regions of memory instead of creating a signature for them with CRC. This moves the overhead from the time dimension to the space dimension. The other technique is to use the virtual memory paging system and its protection mechanisms. In this second case, we would also be able to identify the faulty instruction precisely.

In our current scheme, we detect only active modifications of data. If the data overwritten is identical to the one previously present, we are not able to detect it. Moreover, read accesses are never detected. While read accesses might be allowed in a larger number of cases due to possible processor-level optimizations, it would be useful to be able to detect them. By combining the virtual memory paging system and its protection mechanisms, we believe we can achieve this goal.

7. ACKNOWLEDGEMENTS

We would like to thank Dr. Sayantan Chakravorty for his precious comments while preparing this paper, and David Kunzman for his help with graphics. This work was supported in part by grants NSF 0720827 and NSF ITR-0205611.

8. REFERENCES

[1] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[2] E. Bohm, G. J. Martyna, A. Bhatele, S. Kumar, L. V. Kale, J. A. Gunnels, and M. E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[3] Intel Corporation. Intel Thread Checker. http://www.intel.com.

[4] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[5] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[6] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[7] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[8] S. Kumar. *Optimizing Communication for Massively Parallel Processing*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[9] M. Ronsse and K. D. Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Automated and Algorithmic Debugging*, 2000.

[10] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 34–41, New York, NY, USA, 2006. ACM.

[11] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[12] T. Technologies. *Debugging Memory Problems Using TotalView Debugger*. http://www.totalviewtech.com.

[13] T. Technologies. TotalView® debugger. http://www.totalviewtech.com/TotalView.

[14] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.