FLORIDA STATE UNIVERSITY

MASTER'S PROJECT

Phoenix: A Checkpointing Tool

Author:
Steven ROHR

Supervisor:
Dr. Michael MASCAGNI

A project submitted in partial fulfillment of the requirements for the degree of Master of Computer Science.

January 10, 2017

Steven Rohr defended this project on November 30th, 2016.
The members of the supervisory committee were:

Dr. Michael MASCAGNI
Major Professor

Dr. Sonia HAIDUC
Committee Member

Dr. David WHALLEY
Committee Member

The Computer Science Department has verified and approved the above-named committee members, and certifies that the project has been approved in accordance with university requirements.

Florida State University

Abstract

Dr. Michael Mascagni

Department of Computer Science

Master of Computer Science

Phoenix: A Checkpointing Tool

by Steven ROHR

The programmer must always strive to be mindful of robustness when programming, so as to avoid a crash midway through execution which, in the worst case scenario, would result in a loss of all work that was being done, and require the process to be started over all-together. While many programs have a short enough execution time that they do not need to be concerned with external errors, such as power failure, that is not always the case. In cases where execution time lasts minutes, hours or even days, the programmer may be more concerned with issues such as sudden power loss which would cause unavoidable crashes.

Checkpointing is one way to deal with such unavoidable problems, and typically is done by periodically saving program state so that a restore is possible in case of a crash. Phoenix is a pragma-driven checkpointing tool to simplify the process of creating and restoring from checkpoints for C++ programs, including those that make use of OpenMP for threading.

Acknowledgements

I would like to thank Dr. Mascagni for his ideas and input throughout the course of this project.

I would like to thank my committee members, Dr. Sonia Haiduc and Dr. David Whalley, for their involvement.

I'd like to thank Sharanya Jayarama and Preston Hamlin for their advice on all matters graduate school and I'd like to thank all my friends for all their service as rubber-duck debuggers.

I'd like to thank our Computer Science department for fostering my interest in CS, and for the teaching assistantship which funded my graduate studies.

Contents

Chapter 1

Background & Motivations

1.1 About the Problem

Often times execution time is so short for a program that we worry little about crashes. After all, if an entire run only takes a few seconds, there is little harm in restarting execution should something go wrong. While we are concerned with errors within the program, we often simply choose to ignore external ones.

For some programs, however, execution time may be on the order of minutes, hours or even days. In these circumstances, simply relaunching the program in the event of a crash is less palatable and may even be associated with a fairly hefty monetary cost. Thus, when working with programs with lengthy runtimes, checkpointing, or periodically saving program state, becomes a logical investment.

Methods of checkpointing vary, ranging from backing up the entire program stack to just saving the values of a handful of variables from which the rest could be reconstructed. As is often the case, when presented with two options, there are trade-offs to consider. Checkpointing the entire stack comes with more certainty that one could successfully restore and resume execution. However it is space inefficient in terms of backup size and often time inefficient in terms of how much backup must be written. On the other hand, checkpointing only a few variables can be faster and take up a smaller space-footprint. However the user must be confident that they can reliably reconstruct the program state from only those checkpointed variables.

Regardless of the chosen method in which the checkpointing is done, both of the above have another downside: They require more work on behalf of the programmer, introducing more regions of code which must be debugged. Worse still, the debugging would have to involve deliberately crashing the program. For that reason, those who want to perform checkpointing may benefit from using an existing checkpointing tool where the checkpoint operations have been designed in such a way that they are as atomic as possible and have already been tested in these circumstances.

1.2 Motivations

Checkpointing tools do exist, and so this would not be the first to be written. However, we specifically were interested in providing checkpointing capabilities to parallel programs, specifically in the context of numerical or stochastic methods. Such methods are often highly iterative in nature and require fairly little information to express the current program state. For that reason, checkpointing the entire program stack would often be wasteful in such cases, and we were thus less inclined to use checkpointing tools which do so, rather than checkpointing only a handful of user-specified variables.

As mentioned above, we were also interested in checkpointing parallel programs, and specifically we wanted a way to do so which does not involve dramatically changing the structure of the parallel code by forcing a merge before backup or restore.

For the above reasons, while some checkpointing tools exist for use with C++, and even OpenMP (Greg Bronevetsky, 2004), we decided to create Phoenix, which is ultimately a tool geared towards a fairly niche scenario. Due to the general way it has been implemented, Phoenix could be used to checkpoint a wide variety of C++ applications, regardless of whether or not they are stochastic or numerical in nature.

Chapter 2

A Brief Survey of Existing Tools

2.1 Berkeley Lab's Linux Checkpoint and Restart

Berkeley Lab's Linux Checkpoint and Restart (BLCR) is a kernel-level tool intended to checkpoint applications, and originally designed to be tailored towards use with MPI applications. While Phoenix and many other checkpointing tools utilize system calls to modify the program, BLCR checkpoints from the kernel, allowing the user to checkpoint programs without modifying them in any way. Furthermore, by checkpointing the underlying data structures from within the kernel, BCLR is able to checkpoint data such as process IDs, and can simply reload program state by reloading saved structures at the kernel level. That said, BCLR does not checkpoint everything, as certain structures, such as TCP and UDP sockets, simply proved too difficult to accurately checkpoint.

Beyond the several structures BCLR does not support, BCLR's primary downfall comes into play when checkpointing particularly large applications. In particular, the authors note that BCLR can fall victim to thrashing when checkpointing particularly large applications, which suggests that tools which only checkpoint a portion of the program state may be a better choice in such circumstances (Duell, 2005).

2.2 DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop

DMTCP is a user-level checkpointing tool with support for a wide variety of constructs, such as sockets, fork, ssh, semaphores, shared memory and more. Unlike the previously discussed BLCR, this tool runs at user-level privileges and in the user-space rather than the kernel-space. Similar to BCLR, users do not modify their programs when using DMTCP. Instead, users invoke their programs through DMTCP, allowing it to perform library injection and spawn checkpointing threads for each user-process spawned by the program being checkpointed. Via overloads provided from the injected libraries, DMTCP redirects various function and system calls to

variants of them which have checkpointing wrapper code about them. The downside to checkpointing in this manner, of course, is that there will be some degree of performance hit, both with respect to memory and time (Jason Ansel, 2007).

2.3 Application-level Checkpointing for OpenMP Programs

Like Phoenix, the Application-level Checkpointing tool for OpenMP Programs from Cornell is a compiler-level checkpointing tool which produces self-checkpointing applications by modifying the compilation process. Also like Phoenix, the Cornell Checkpointing Compiler (CCC) focuses primarily on OpenMP applications. CCC argues against having the programmer make dramatic edits, but acknowledges that simply checkpointing everything all the time is an inefficient approach. In order to use CCC, the user periodically make function calls to a provided function, ccc.potential.checkpoint. This function will then determine what needs to be checkpointed and whether or not enough time has elapsed to justify the creation of a checkpoint (Greg Bronevetsky, 2004).

Chapter 3

The Idea & Constraints

3.1 Phoenix: An Introduction

As we mentioned before, there are two immediately apparent and logical approaches to checkpointing a program. The first is to checkpoint the entire stack-state. While this method is surely bound to accurately capture a snapshot of the program, and restoring from such a snapshot should absolutely be feasible, it is also by far the more complicated method, and would also result in fairly large checkpoints. Thus, we decided to go with the latter discussed method of checkpointing; users define specific variables to checkpoint, and nothing else will be checkpointed.

Seeing as Phoenix needs to allow the user to designate what needs be checkpointed and when, we needed a way for the programmer to signal these things to Phoenix. We chose to emulate OpenMP's pragma-driven nature, and offer several pragmas which will be discussed later in this document. The program Phoenix operates during compilation-phase on a given program which contains Phoenix pragmas, modifying the code with file I/O statements, conditionals and the occasional carefully placed goto statement in accordance with the pragmas put in place by the programmer.

The result is an augmented variant of the original program which now performs periodic backups. Should it find a checkpoint during execution, the program will restore variable state from that checkpoint and fast-forward to the line at which the checkpoint was created. In this way, we can provide a set of generic checkpointing tools which the user can then apply in a number of different ways to their program to satisfy their individual checkpointing needs.

3.2 Project Constraints

3.2.1 C++, not C

As aforementioned, Phoenix performs checkpoints via file I/O. We chose this as files are perhaps the simplest way to store persistent data. From there, we had to decide how this I/O would be performed. Ultimately, the extensiblity of C++'s insertion and extraction operators (“«" and “»") was decided to be highly useful to Phoenix, as now the user need not inform Phoenix at all what type variable they are checkpointing, they need only be sure that the variable's type has appropriate fstream insertion and extraction overloads. The user should be careful to ensure that any overloads to the extraction operator prints data in the same format which the overloaded insertion operator reads it in, else Phoenix will most assuredly not function as intended.

3.2.2 Compilation Phase Metaprogramming

Phoenix runs in the compilation phase for a given program, meaning that any code it inserts into the source program must be fairly static. This, to some degree, does restrict the kind of behavior which is even possible for Phoenix. For instance, it is not possible for a Phoenix-augmented program to alter the contents of a label during run-time. That said, any software design comes with inherent limitations, and at this time we have not found any compelling features which Phoenix cannot have as a compilation-phase metaprogramming tool, and so we posit that Phoenix's general design suits its purpose.

3.2.3 One Checkpoint per Label (and Thread)

Originally, Phoenix planned to allow the user to define the number of checkpoints to be kept and they would be differentiated via timestamp. Upon needing to restore, the program would find the most recent checkpoint by comparing timestamps and begin by attempting to restore from that checkpoint, going down the list should there be an issue.

This behavior was scrapped for two reasons. Firstly, C++ does not have a native way to access directory listings without making use of ls via system call or exec. Boost does support this, but we were hesitant to force Boost as a dependency for any program augmented by Phoenix. This meant Phoenix would have to metaprogram code which parses the output of ls (or dir, so that we can support Windows too). Secondly, we decided that the cases where a checkpoint would be corrupted are few and far between, and such cases may also be indicative of far more serious issues which Phoenix alone would not be able to safeguard against. Therefore, we decided

that supporting multiple checkpoints is not currently a compelling enough feature to warrant its execution.

3.2.4 #pragma omp parallel for

Omp paralel for is used to parallelize a for loop using OpenMP. Essentially, it partitions the loop iterations, handing off a roughly equal number of iterations to each of several threads. Needless to say, this is a useful feature of OpenMP, and regretfully we cannot support checkpointing loops which have been parallelized in this way.

The dilemma comes from the method by which OpenMP parallelizes these loops. The inner body of the loop is effectively converted to a function, and the loop body is replaced with a sequence which spawns a thread, then calls the above-mentioned function (Yliluoma, 2007). The issue arises with the fact that the loop iterator is then unmodifiable within the loop body. As we will discuss later in the paper, Phoenix restores loops by dedicating their first iteration to restoring from the backup, then resumes execution as normal. Seeing as OpenMP does not allow the body of a parallelized for loop to modify the iterator, restoring such a loop with Phoenix's current loop restore methodology is simply not possible.

3.2.5 Semantic Errors

As we have already mentioned, Phoenix is a metalanguage of pragmas. As with any other method of code writing or generation, there is an inherent possibility of user created semantic errors, which would result in the program not behaving as intended. While we attempt to make the pragmas as simple to use as possible, there exists a real chance that users will misuse Phoenix's pragma constructs and as a result their program will misbehave. This shortcoming could be mitigated to some degree by having Phoenix attempt to identify various common errors and warn the user of them, however as is often the case with code, it is hard to determine what is intentional rather than accidental. Thus, the best way to prevent such semantic errors is for the user to read the documentation and invoke Phoenix thoughtfully.

Chapter 4

Methodology

4.1 Code Design

Phoenix is a compilation-phase parser with metaprogramming capability. It accepts another program's source code as input, scans it, and transcribes a new file of augmented source code. The vast majority of the new file is the same as the original, barring the fact that any pragmas directed at Phoenix have been replaced with checkpointing code.

Phoenix makes two passes on a given input program. The first pass is dedicated to finding which variables the user wants to checkpoint. These variable names are added to a vector of sets, where each set is associated with one checkpoint label (this process will be more comprehensively defined in Chapter 4). The second pass actually metaprograms the new file, transcribing most of the code from the input unchanged, but adding in checkpointing code as directed by the pragmas.

As to code-structure, Phoenix uses a fairly standard design where the main control loop reads the input program one line at a time, using regexes to determine if the line is one of Phoenix's pragma directives. If it is, a corresponding function is called to generate any code that would be required here, which is then returned as a String object to be concatenated to the end of the new program. If the line in question did not match any regex, the original line is concatenated to the end of the program unmodified. This structure makes adding new pragma directives to Phoenix fairly simple, as the author need only define a regular expression to identify such directives and a String accepting and returning function to perform any metaprogramming.

4.2 Language

While Phoenix operates on C++ code, we chose to write it in Java due to Java's superior string parsing and manipulation libraries, including native regex support

(which we use to identify pragmas directed at Phoenix as well as certain code constructs), a string split method and more. Phoenix also makes use of Java's built-in templated library and highly object-oriented nature using a variety of classes to manage information gleaned from pragma directives aimed at Phoenix, such as which variables need be checkpointed and when.

4.3 Invoking Phoenix

At this time, Phoenix is invoked via manually calling the java program PHOENIX. The user passes a single parameter, being the input file, and Phoenix then generates a new source code file (name being the base filename with "_CHECKPOINTING_ENABLED" appended before the extension). The user may compile this new file with g++ to obtain a variant of their program which has Phoenix's modifications in place. Ideally Phoenix would be invoked via g++ through use of a compiler flag, however we were not able to implement that feature due to time constraints.

Chapter 5

Definitions

5.1 Definitions

5.1.1 Label

While discussing the features of Phoenix and how to use it, the term “label" will show up a number of times. In Phoenix, a label refers to a user-given name of a collection of variables which is associated with a given checkpoint. Essentially, users define labels as collections of variables and checkpoint the label, not the variables. This results in Phoenix keeping a number of checkpoints, one per label and per thread. Said checkpoints are kept within a folder which a Phoenix-augmented program will create and destroy as need be.

This system allows the checkpoint pragmas to be more legible and easier to manage. Rather than specifying that you want to checkpoint the variables “lowerGuess", “upperGuess", “errorEstimate" and perhaps several others, a user could just say that they want to checkpoint the label “Bisection_Method", where they had previously defined that label to contain those several variables.

As we mentioned briefly earlier, Phoenix also makes use of goto statements. This is what allows us to “jump" across a region of code encapsulated by pragma statements that define the start and end of a checkpointed region. As the label name doubles as the target for the goto statement associated with that label, any given label may only be checkpointed once for any given program so to prevent ambiguity during a jump.

5.1.2 Metaprogramming

The phrase metaprogramming appears multiple times throughout this document. A metaprogram is a term for a program which operates on another program, either by accepting a program as input or producing one as output. Phoenix does both, and thus would be considered a metaprogram.

Chapter 6

The Pragmas

6.1 Label Declaration & Modification

6.1.1 #pragma PHOENIX_VAR {LABELS} {VARIABLES}

PHOENIX_VAR is used to define new labels or add more variables to an existing label. Both the “{LABELS}" and “{VARIABLES}" fields require the curly braces, and both allow multiple label and variable names to be listed, which should be whitespace delimited. In this way, the user can add multiple variables to one or more labels at once with a single pragma line.

As we mentioned before, Phoenix makes two passes on a given input program; the first pass is almost entirely focused on finding lines of this pragma and constructing the labels as the user has defined them. Upon finding a line with this pragma, Phoenix first checks if it has previously seen this label name. If it has, Phoenix adds any variables listed to the set of variable objects associated with this label. If it has not, Phoenix first creates a new label object, then adds the variables to the inner set and the label object to a list of label objects.

As we are using sets for the variables, the user does not need to worry about accidentally adding multiple copies of a single variable to a single label. As this happens in a pass where no code manipulation occurs (and no code is generated ever at the location of this pragma), PHOENIX_VAR can be placed anywhere, even outside any function body where the variables it mentions are not declared. When actually checkpointing the variables, we simply write or read them to/from the file in the order in which they are stored in the set. As nothing can be added or removed from the set between writing to a checkpoint and reading from it, we can safely assume the ordering will not change.

{VARIABLES}

The variables field takes any number of variable names, whitespace delimited. As we mentioned in chapter 2, we make use of C++'s insertion and extraction operators, allowing near any variable type (assuming it has the appropriate operator overloads). We also allow arrays of any type with insertion and extraction fstream operators, and provide a specific syntax in how they should be added to a label:

$$[name \;\; number\_elements \;\; starting\_index]$$

Name is the name of the array variables, number_elements is how many elements should be checkpointed, and starting_index is an optional parameter detailing the first index to checkpoint (defaulting to 0 if not given). Both number_elements and starting_index may be numbers or variable names, as they are stored as Java String objects within Phoenix and transcribed as entered into a for loop which checkpoints the array. The user should be careful, however, with using variables here and should ensure that those variables would already have a value (in case they are also being restored) upon reaching this point in the restore-code.

There is no pragma construct to remove a variable from a label. If the user is asking why that is, they should remember that Phoenix runs entirely during compilation phase, and after which their (modified) code runs alone. Combine that with the fact that each label may only be checkpointed once, and the idea of removing variables from a checkpoint makes little to no sense. Instead, users should just modify their PHOENIX_VAR pragma statement to never include the offending variable in the first place.

6.2 Starting a Checkpoint

6.2.1 #pragma PHOENIX_START LABEL

Checkpoints in Phoenix are best thought of as regions, which can be associated with a name (being their label). This pragma allows the user to define the start of such a region, at which all variables being checkpointed or otherwise used within this label should already be declared. This is the location in which the modified program where the program will check if a valid restore file exists, and if such a file exists, this is where the goto statement will jump from, with the target being the location of the actual backup portion of the checkpoint.

Due to safety restrictions within goto, PHOENIX_START must be in the same level of scope as the actual checkpoint pragma. This restriction can be circumvented with certain g++ compiler flags, however we do not advise doing so as jumping across scope can result in all sorts of especially absurd behavior.

6.3 Kinds of Checkpoints

After starting a label, users need also actually denote the location of the code (and potentially frequency) of when the checkpointing should actually occur. Users do that by pairing PHOENIX_START with one of the four below pragmas.

6.3.1 #pragma PHOENIX_CHECKPOINT LABEL

PHOENIX_CHECKPOINT is the simplest form of checkpoint. It is unconditional, for single-threaded regions and happens once each time the program reaches this line. It accepts only one parameter: the name of the label to checkpoint. This pragma would most often be used to wrap an intensive region of code, checkpointing any important results after the calculations have occurred so to save time in future runs of the program.

How it works

Phoenix enters an if/else sequence in the resultant program at the location of this pragma. The if statement handles case restore, and has an impossible condition (false). Within that if statement is the goto target for this checkpoint, thus the only way to reach the restore code is when a previous checkpoint is detected at the start pragma. The else case is the backup sequence, which simply writes the variables associated with the label to the file.

6.3.2 #pragma PHOENIX_LOOP LABEL CONDITION

PHOENIX_LOOP is intended for use within for, while and do while loops, and is expected to be placed on the line before the loop begins. As with all other checkpoint pragmas offered by Phoenix, the user must designate which label this checkpoint is associated with. The CONDITION parameter is taken to be everything after the label name remaining on this line, and will be used verbatim within the checkpointed loop to determine which cases should checkpoint their state (alongside normal operations).

How it works

At the immediate location of the pragma, Phoenix writes an unreachable if statement containing the goto target for this checkpoint and a line which sets a boolean flag to true which signals the need to restore from a file. Within the below loop, Phoenix adds two if statements.

The first if statement checks if the aforementioned boolean flag was set, and if so, restores from the file before unsetting the flag. The loop then continues as normal. As the restore boolean is only set before the loop begins, it is only possible to do a restore on the first iteration of a loop. It is worth noting that unless the user has added the loop iterator to the label it will not be checkpointed, which will likely result in unintentional behavior.

The second if statement checks if this is an iteration where we should update the checkpoint. If it is, the checkpoint file is opened and updated. Regardless of the status of the above two if statements, the user's original loop body then executes as originally written.

6.3.3 #pragma PHOENIX_OMP LABEL

PHOENIX_OMP is a specialized variant of PHOENIX_CHECKPOINT which is for use within non-loop OpenMP parallel regions. It essentially behaves the same as PHOENIX_CHECKPOINT, however it creates multiple checkpoints, one per thread. Each of those checkpoints is labeled with its creator's threadID (obtained via omp_-get_thread_num()). This allows us to ensure that in case restore each thread gets the data it originally had, which may be important for computational reproducibility, especially in cases where there are cross-thread operations (such as parallel reduction).

PHOENIX_OMP also makes use of an OpenMP barrier sync at the end of all backups and restores, which will force any threads to wait until all threads have completed the operation before proceeding. This ensures that any cross-thread operations will only happen after all threads have restored to the state they were in before, helping avoid the strange behavior or race conditions programmers who work with parallelism are all too aware of.

6.4 #pragma PHOENIX_PARLOOP LABEL CONDITION

PHOENIX_OMP is to PHOENIX_CHECKPOINT as PHOENIX_PARLOOP is to PHOENIX_LOOP. That is to say, this is the variant of loop checkpointing to be used when checkpointing a loop within a OpenMP parallel region. Like PHOENIX_OMP, it makes use of threadIDs in order to ensure each thread receives the correct data on case restore, but otherwise it behaves very much like PHOENIX_LOOP. Also like PHOENIX_OMP, this pragma makes use of OpenMP's barrier sync directive to ensure no thread races ahead of the others in the event that we are doing a backup or restore.

As we mentioned before, this checkpoint regrettably cannot work with a loop parallelized through "#pragma omp parallel for" due to how OpenMP parallelizes

loops (Yliluoma, 2007). If our users would like to checkpoint such a region of code, they are encouraged to either refactor their loop to be a loop within a parallel region or simply make use of a normal PHOENIX_OMP after the loop body.

6.5 Other pragmas

6.5.1 #pragma PHOENIX_CLEANUP

PHOENIX_CLEANUP is to be used at the end of a user's program and serves to remove all traces of checkpoints so that the user does not have to do so manually. At the location of this pragma, Phoenix adds a call to the system() function which removes the checkpoint directory and all files within it. As this pragma operates on the entire folder of checkpoints and is entirely undiscerning in what it removes, we suggest it be placed at the end of the program such that it only runs in the case that the program finished execution. That said, this pragma is entirely optional and the user is welcome to manually manipulate checkpoints, perhaps by adding such functionality to a makefile.

Chapter 7

Conclusions

7.1 Future Work

The author of this paper has worked on Phoenix for approximately one year at this point, throughout the latter half of their time as a graduate student. While Phoenix does do a lot of what we initially wanted, it is not complete, and as this author is preparing to graduate, there is some question of where the project will go from here. This section will discuss features core to Phoenix's original goal which still require implementation, and offer suggestions in how this author envisions they may be implemented.

7.1.1 g++ Integration

As Phoenix was intended to emulate OpenMP's pragma-driven nature, it was also intended to emulate how easily OpenMP may be invoked via g++. As mentioned earlier in this paper, we ultimately did not find time to implement this feature, however we posit that adding a new compiler flag to g++ (or some other compiler) would not be all too difficult with more research. Preferably this would be implemented via some kind of install script for Phoenix, which could also copy any other future internal Phoenix files to their destinations.

7.1.2 Further Refactoring for Further Extensibility

Earlier in Chapter 3 we discussed the current state of Phoenix's codebase. As a refresher, Phoenix scans for labels via heavy use of regular expressions, and upon encountering a pragma aimed at it, calls metaprogramming functions with the current line of text (and potentially also the scanner object) so to create checkpointing code.

While this is a fairly extensible system, and this author would not suggest dramatically changing it, Phoenix would benefit further by being broken up into more files in such a way that future maintainers could add more pragma directives by

adding another file which contains the regex and metaprogramming function, then making a minor modification to the central parser to include their new pragma.

7.1.3 Keeping Multiple Checkpoints

As we have previously discussed, keeping multiple checkpoints would serve to provide further resilience in the case that the most recent checkpoint it corrupted. An example case of that may be if power loss occurs during the writing of a checkpoint. Perhaps the simplest way to handle this is to keep only a set number, perhaps the most recent two or three, and attempt to restore from a previous checkpoint if the most recent is corrupt. Of course, this adds another requirement: Actually determining if a checkpoint is corrupt. That feature is probably far less trivial, perhaps to the scale of being an entire other project or thesis depending on the exact decided scope.

7.1.4 #pragma omp parallel for

This is perhaps one of the trickier pieces of future work, in the sense that Phoenix's current checkpointing model simply does not work for this. While the openMP checkpoints Phoenix currently offers are fairly similar to their single-threaded variants, in order to checkpoint a loop where the iterations have been split over several threads the implementer would need come up with an entirely new method. Perhaps the best way to checkpoint such a loop is by actually forcing a thread-merge, checkpointing, then relaunching the parallel threads, however it is also entirely possible that this method would also not work. Ultimately, this task would likely require the implementer to venture deep into OpenMP's source code and trying a multitude of ideas.

7.1.5 Vision of the End Product

Wherever future maintainers take Phoenix, this author feels they should strive to make using Phoenix fairly simple for the end user. Such users should only need to take a few moments to ask themselves what variables need to be checkpointed, toss some pragma directives into their program, then recompile the modified program. Phoenix should probably stick to effectively being an add-on to compilers rather than some additional process which runs along-side the target program, however this does result in the unfortunate side effect that maintainers need practice metaprogramming to add additional features.

7.2 Proposed Features

While Phoenix is capable already of checkpointing a wide variety of programs and variable types, a number of useful yet not implemented features have been proposed. This section will serve as an overview of those proposals, including some thoughts on feasibility, methodology and likely issues that would be encountered.

7.2.1 Remote Checkpoints

The idea behind remote checkpointing would be that all checkpoints are stored at a third party location, so as to provide even more resilience against catastrophic failure. Such a feature could probably be implemented by making use of execv or system to make calls to scp, where the remote server information (address, needed credentials) is provided to Phoenix through command line arguments in some way.

As the entire purpose of Phoenix is computational resilience, this feature would probably be best implemented with some hash-checking of the remote file to ensure that the file has indeed been correctly offloaded and not corrupted. Furthermore, due to the latency of network operations, offloading checkpoints may be best done in a new thread or process.

7.2.2 Support For More Parallization

As we already support the majority of OpenMP, it would be nice to expand such support to other forms of parallelization such as pthreads, fork or even CUDA. Seeing as we aim to provide thread-wise checkpoints that do not require some kind of merge before backing up or restoring, we would either need to provide a new variant of checkpoint for each form of parallelization which gets the thread (or process) ID in the needed way, or perhaps we would have the user supply us this information via a variable or function call as a pragma argument which, when invoked, would return the thread's identifier.

One major issue with supporting fork, however, is that we would need some other way to enumerate processes as process IDs will not be the same between runs. Perhaps the method would involve making use of pipes, however, metaprogramming with fork and pipes would likely be a challenge like no other. The author of this paper does not envy the poor soul tasked with such an implementation.

7.2.3 Better Error Detection/Reporting

As we have already mentioned, Phoenix is a metaprogramming tool and is thus as prone to semantic errors as any given programming language. We have attempted

to make the pragmas as robust as possible, and as logical as possible in how they operate. That said, we expect the occasional user to misuse Phoenix, and so detection of common errors (not checkpointing loop iterators, attempting to checkpoint omp parallel for, etc), it would be nice for Phoenix to inform the user via warning that they have probably made an error.

That said, Phoenix also includes a readme file which details how to use every pragma it includes, and discusses their limitations as this paper does. Therefore, we argue that while nice, this feature is not necessary. Furthermore, in order to properly compile a list of common misconceptions and errors we would likely need to run a sort of study with a number of Phoenix users, and at this time of this paper's authorship such a user-base simply doesn't exist.

7.3 Conclusion

Phoenix is a compilation-phase checkpointing tool able to checkpoint a wide variety of C++ applications, including ones which contain OpenMP functionality. Furthermore, Phoenix is highly extensible by making use of C++'s own operator overloading paradigm in order to provide checkpointing support for practically any variable type, including those defined by the user. We hope that Phoenix will prove useful to the community, and that it may make dealing with external errors less painful for its users.

Appendix A

Sample Phoenix Program

A.1 Fizz-Buzz

A.1.1 Before Phoenix Operates

```
#include <iostream>
#include <unistd.h>

using namespace std;

int main(){
   #pragma PHOENIX_VAR {checkpoint1} {i}
   #pragma PHOENIX_START checkpoint1
   #pragma PHOENIX_LOOP checkpoint1 i % 3 == 0
   for (int i = 1; i < 200; i++){
      cout << i << ": ";
      if (i%3 == 0)
         cout << "fizz";
      if (i%5 == 0)
         cout << "buzz";
      cout << endl;
   }
   return 0;
}
```

A.1.2 After Phoenix Operates

```
#include <fstream>
#include <cstring>
#include <stdlib.h>
bool PHOENIX_Restore_bool = false;
#include <iostream>
#include <unistd.h>
using namespace std;
int main(){

{
  system("mkdir -p PHOENIX_slow_fizzbuzz/");
  char PHOENIX_slow_fizzbuzz_checkpoint1_cstr [100] = "";
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
      "PHOENIX_slow_fizzbuzz/");
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
      "backup_slow_fizzbuzz_checkpoint1.txt");
  ifstream PHOENIX_slow_fizzbuzz_ifstream_var;
  PHOENIX_slow_fizzbuzz_ifstream_var
      .open(PHOENIX_slow_fizzbuzz_checkpoint1_cstr);
  if (PHOENIX_slow_fizzbuzz_ifstream_var) goto checkpoint1;
}

if (false) {
  checkpoint1:
  PHOENIX_Restore_bool = true;
}
   for (int i = 1; i < 200; i++){
{
if (PHOENIX_Restore_bool){
{
  ifstream PHOENIX_slow_fizzbuzzcheckpoint1_ifstream_var;
  char PHOENIX_slow_fizzbuzz_checkpoint1_cstr [100] = "";
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
      "PHOENIX_slow_fizzbuzz/");
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
      "backup_slow_fizzbuzz_checkpoint1.txt");
  PHOENIX_slow_fizzbuzzcheckpoint1_ifstream_var
      .open(PHOENIX_slow_fizzbuzz_checkpoint1_cstr);
  PHOENIX_slow_fizzbuzzcheckpoint1_ifstream_var >> i;
  PHOENIX_slow_fizzbuzzcheckpoint1_ifstream_var.close();
}
  PHOENIX_Restore_bool = false;
}
}
if (i % 3 == 0) {
```

```
{
  ofstream PHOENIX_slow_fizzbuzzcheckpoint1_ofstream_var;
  char PHOENIX_slow_fizzbuzz_checkpoint1_cstr [100] = "";
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
     "PHOENIX_slow_fizzbuzz/");
  strcat(PHOENIX_slow_fizzbuzz_checkpoint1_cstr,
     "backup_slow_fizzbuzz_checkpoint1.txt");
  PHOENIX_slow_fizzbuzzcheckpoint1_ofstream_var
     .open(PHOENIX_slow_fizzbuzz_checkpoint1_cstr);
  PHOENIX_slow_fizzbuzzcheckpoint1_ofstream_var << i << endl;
  PHOENIX_slow_fizzbuzzcheckpoint1_ofstream_var.close();
}
}
      cout << i << ": ";
      if (i%3 == 0)
         cout << "fizz";
      if (i%5 == 0)
         cout << "buzz";
      cout << endl;
   }

   return 0;
}
```

Appendix B

PHOENIX.java

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.util.*;

public class PHOENIX {
   Labels L = new Labels();

   public final String LINE_LABEL = "\\s*#pragma PHOENIX_VAR.*";
   public final String LINE_START = "\\s*#pragma PHOENIX_START.*";
   public final String LINE_CHECKPOINT = "\\s*#pragma
      PHOENIX_CHECKPOINT.*";
   public final String LINE_LOOP = "\\s*#pragma PHOENIX_LOOP.*";
   public final String LINE_OMP  = "\\s*#pragma PHOENIX_OMP.*";
   public final String LINE_PARLOOP = "\\s*#pragma
      PHOENIX_PARLOOP.*";
   public final String LINE_CLEANUP = "\\s*#pragma
      PHOENIX_CLEANUP.*";

   public final String PHOENIX_INCLUDES = "#include <fstream>\n"
                                        + "#include <cstring>\n"
                                        + "#include <stdlib.h>\n"
                                        + "bool PHOENIX_Restore_bool =
                                            false;\n"
         ;

   String basefilename = null;

   public static void main(String [] args){
      PHOENIX phoenix = new PHOENIX();
      String infile = args[0];

      File input = new File(infile);
      phoenix.basefilename = (input.getName().split("\\."))[0];
```

```
        File output = new File(phoenix.basefilename +
            "_CHECKPOINTING_ENABLED.cpp");

        //Gather labels, store them in program memory.
        //No edits to file made yet.
        phoenix.LabelScanner(input);

        //Create the metaprogrammed file with checkpointing code.
        phoenix.MetaprogramFile(input, output);

        System.out.println(output.getName());

    }

    public boolean LabelScanner(File cppProgram){
        //Initial phase of checkpointer. Gathers list of labels to
            use in checkpoints and denotes which
        //variables belong in each label (all user defined).

        boolean found_start = false;

        try {
            //Gather labels
            Scanner sc = new Scanner(cppProgram);
            while (sc.hasNext()) {
                String line = sc.nextLine();
                if (line.matches(LINE_LABEL)){
                    L.AddLine(line);
                }
            }

            //Mark OMP labels, mark labels with a start statement
            sc = new Scanner(cppProgram);
            while (sc.hasNext()) {
                String line = sc.nextLine();
                if (line.matches(LINE_START)){
                    String [] s = line.trim().split("\\s+");
                    String label = s[2];
                    for (Labels.Label l : L.vLabels){
                        if (l.name.equals(label)){
                            l.started = true;

                            if (!found_start) {
                                l.firstStarted = true;
                                found_start = true;
                            }
                            break;
                        }
```

```
            }
         }
         if (line.matches(LINE_OMP) ||
            line.matches(LINE_PARLOOP)){
            String [] s = line.trim().split("\\s+");
            String label = s[2];
            for (Labels.Label l : L.vLabels) {
               if (l.name.equals(label)) {
                  l.isOMP = true;
                  break;
               }
            }
         }
      }

      return true;
   } catch (FileNotFoundException e) {
      e.printStackTrace();
      return false;
   }
}

public boolean MetaprogramFile(File input, File output){
   try {
      //Begin a new file.
      output.createNewFile();

      //Declare IO tools
      Scanner sc = new Scanner(input);
      BufferedWriter fw = new BufferedWriter(new
         FileWriter(output));

      //Include some libraries we will be needing.
      fw.write(PHOENIX_INCLUDES);

      while (sc.hasNext()) {
         String line = sc.nextLine();
         String addline = null;

         if (line.matches(LINE_LABEL)) {
            //Do nothing.
         }
         else if (line.matches(LINE_START)){
            addline = MP_START(line);
         }
         else if (line.matches(LINE_CHECKPOINT)){
            addline = MP_CHECKPOINT(line, sc);
         }
```

```
            else if (line.matches(LINE_LOOP)){
               addline = MP_CHECKPOINT_LOOP(line, sc);
            }
            else if (line.matches(LINE_OMP)){
               addline = MP_CHECKPOINT_OMP(line, sc);
            }
            else if (line.matches(LINE_PARLOOP)){
               addline = MP_CHECKPOINT_OMP_LOOP(line, sc);
            }
            else if (line.matches(LINE_CLEANUP)){
               addline = MP_KILLBACKUPS(line);
            }
            else{
               //Not one of our things. Transcribe it faithfully.
               addline = line;

            }

            if (addline != null && !addline.trim().equals("")) {
               fw.write(addline);
               fw.newLine();
            }

         }

         fw.close();

      } catch (Exception e) {
         e.printStackTrace();
         return false;
      }

      return false;
   }

   Labels.Label getLabel (String s){
      Labels.Label lptr = null;

      for (Labels.Label l : L.vLabels){
         if (l.name.equals(s)){
            lptr = l;
            break;
         }
      }
      return lptr;
   }

   String GetCstringName(String fname, String label){
```

```
    return "PHOENIX_" + fname + "_" + label + "_cstr";
}

String GetDirectoryName(String fname){
    return "PHOENIX_" + fname + "/";
    //return "";
}

//Notice to the reader/future maintainers:
//Functions preceeded by "MP_" return lines of C++ code to be
    inserted in the resultant output file.
//MP stands for metaprogramming.

String MP_Declare_and_strcat_Backup(String label, boolean omp,
    String fname){
    String ret = null;
    fname.replace(".", "_");
    if (!omp) {
        ret = "\tchar " + GetCstringName(fname, label) + " [100]
            = \"\";\n"
                + "\tstrcat(" + GetCstringName(fname, label) + ",
                    \"" + GetDirectoryName(fname) + "\");\n"
                + "\tstrcat(" + GetCstringName(fname, label) + ",
                    \"" + "backup_" + fname + "_" + label + ".txt" +
                    "\");\n";
    }
    else{
        ret = "\tchar " + GetCstringName(fname, label) + " [100]
            = \"\";\n"
                + "\tstrcat(" + GetCstringName(fname, label) + ",
                    \"" + GetDirectoryName(fname) + "\");\n"
                + "\tstrcat(" + GetCstringName(fname, label) + ",
                    \"" + "backup_" + fname + "_" + label + "_\");\n"
                //cstring = cstring + "_" + omp_thread_num + ".txt"
                + "\tsprintf(" + GetCstringName(fname, label) + ",
                    \"%s%d.txt\", " + GetCstringName(fname, label) +
                    ", omp_get_thread_num());\n";
    }

    return ret;
}

public String MP_START(String line){
    //Generate code to jump to restorepoint.
    String ret = null;

    //Format: #pragma PHOENIX_START <LABEL>
    String [] s = line.trim().split("\\s+");
```

```
        String label = s[2];

        Labels.Label lptr = getLabel(label);

        if (lptr != null) { //This label exists
            //Check if there exists a backup file for this label
                (case restore)


            String backupName = GetCstringName(basefilename,
                lptr.name);
            String ifstreamvar = "PHOENIX_" + basefilename +
                "_ifstream_var";
            ret = "\n{\n";
            if (lptr.firstStarted)
                ret += "\tsystem(\"mkdir -p " +
                    GetDirectoryName(basefilename) + "\");\n";


            ret += MP_Declare_and_strcat_Backup(lptr.name,
                lptr.isOMP, basefilename)
                + "\tstd::ifstream " + ifstreamvar + ";\n"
                + "\t" + ifstreamvar + ".open(" + backupName + ");\n"
                + "\tif (" + ifstreamvar + ") " + "goto " + label + ";"
                + "\n}\n"
            ;
        }

        return ret;
    }

    public String MP_KILLBACKUPS(String line){
        return "\tsystem(\"rm -rf " + GetDirectoryName(basefilename)
            + "\");\n";
    }

    public String MP_CHECKPOINT(String line, Scanner sc){
        //Parse the pragma line and generate the appropriate
            checkpoint code
        String ret = null;

        //Format: #pragma PHOENIX_CHECKPOINT <LABEL>
        String [] s = line.trim().split("\\s+");
        String label = s[2];

        Labels.Label lptr = getLabel(label);

        if (lptr != null) { //This label exists
```

```
        //Checkpointing is an if/else.
        //The if is the restore case and should be unreachable
            except with goto.
        String ifstreamvar = "PHOENIX_" + basefilename +
            lptr.name + "_ifstream_var";
        String ofstreamvar = "PHOENIX_" + basefilename +
            lptr.name + "_ofstream_var";

        ret = "if (false) {\n" //Impossible. Thus only reachable
            through goto label.
            + "\t" + lptr.name + ":\n"
            + lptr.MP_RestoreLabel(ifstreamvar)
            + "\t;\n"
            + "}\n"
        ;
        //The else is the backup case and happens whenever we
            don't restore.
        ret += "else {\n"
            + lptr.MP_BackupLabel(ofstreamvar)
            + "}\n"
        ;
    }

    return ret;
}

public String MP_CHECKPOINT_LOOP(String line, Scanner sc){
    //Special case of MP_CHECKPOINT in which we checkpoint a
        normal loop
    //This pragma is placed directly above the loop.
    //So we can count curly braces to determine loop body
        (important, determines when to checkpoint).

    String ret = null;

    //Format of this line: #pragma PHOENIX_LOOP <LABEL>
        <CONDITION>
    String [] s = line.trim().split("\\s+", 4); //Everything
        after label is CONDITION
    String label = s[2];
    String CONDITION = s[3]; //If it doesn't have a 3 their code
        is broken.

    Labels.Label lptr = getLabel(label);


    if (lptr != null) { //This label exists
        //Need target for goto on restore
```

```
String macroname = "PHOENIX_LOOP_" + lptr.name;
String ifstreamvar = "PHOENIX_" + basefilename +
    lptr.name + "_ifstream_var";
String ofstreamvar = "PHOENIX_" + basefilename +
    lptr.name + "_ofstream_var";

ret = "PHOENIX_Restore_bool = false;\n"
   + "if (false) {\n"
   + "\t" + lptr.name + ":\n" //Label for goto
   + "\tPHOENIX_Restore_bool = true;\n"
   + "}\n"
;

//Need to find the loop body now, that's where
    checkpointing code occurs.
//We find loop body by scanning lines to find the loop
    (for, while or do while),
//then by counting curly braces.
//Hard bit is we need to handle many cases.

//Case isOMP we must also find the omp pragma and ensure
    the restore bool is PRIVATE.
boolean loopHadCurlyBrace = false;
boolean loopOpen = false;
int curlyOpenCount = 0;
String remainingCode = "";

while (sc.hasNext() && !(loopOpen && curlyOpenCount > 0))
    {
   String l = sc.nextLine();
   //Case OMP -
   //Check if this is pragma line.
   //If so, make the bool private.
   if (lptr.isOMP &&
       l.matches("\\s*#pragma\\s+omp\\s+(for|parallel).*")){
      //Already has private field
      if (l.matches(".*firstprivate\\s*(.*).*")){
         String [] sta =
             l.split("firstprivate\\s*\\Q(\\E", 2);
         ret += sta[0]
            + " "
            + "firstprivate("
            + "PHOENIX_Restore_bool, "
            + sta[1]
            + "\n"
         ;
      }
      //No private field, make one
```

```
            else{
                String [] sta = l.split("//", 2);
                ret += sta[0]
                    + " "
                    + "firstprivate(PHOENIX_Restore_bool)"
                ;

                if (sta.length > 1 && !sta[1].equals("")){
                    ret += "//"
                        + sta[1]
                    ;
                }

                ret += "\n";
            }
        }

        //Case loop not yet open
        else if (curlyOpenCount == 0) {
            //Case open curly on same line.
            if (l.matches("\\s*for.*\\Q{\\E.*") ||
                l.matches("\\s*while.*\\Q{\\E.*") ||
                l.matches("\\s*do.*\\Q{\\E.*")) {
                ret += l + "\n";
                loopOpen = true;
                loopHadCurlyBrace = true;
                curlyOpenCount++;
            }
            //Case loop start without open curly on same line.
                Still need to find open curly brace.
            //Logic dictates it's the next line.
            else if (l.matches("\\s*for.*") ||
                    l.matches("\\s*while.*") ||
                    l.matches("\\s*do.*")) {
                ret += l + "\n";;
                loopOpen = true;
            }
            //Case curly brace, maybe followed by other stuff.
            else if (l.matches("\\s*\\Q{\\E*.")) {
                String[] stuff = l.split("\\Q{\\E", 2);
                curlyOpenCount++;
                loopHadCurlyBrace = true;
                ret += "\t{\n";
                if (stuff.length == 2) //There is stuff after
                    the curly brace. Who puts stuff there?!
                    remainingCode += stuff[1] + "\n";;
            }
            //Case anything else.
```

```
                //Why aren't they starting the loop yet?! We asked
                    it be the line after the pragma!
                else {
                    ret += l + "\n";
                }
            }
        }

        //Add the restore sequence here.
        ret += "if (PHOENIX_Restore_bool){\n";
        ret += lptr.MP_RestoreLabel(ifstreamvar)
            + "\tPHOENIX_Restore_bool = false;\n"
            + "}\n";

        //Now add the backup sequence
        ret += "if (" + CONDITION + ") {\n"
            + lptr.MP_BackupLabel(ofstreamvar)
            + "}\n"
        ;

        //Add back any code that idiot amateurgrammer put
            directly after the curly brace.
        ret += remainingCode;

        //Rest of loop. Just add it I guess.
        while (sc.hasNext() && curlyOpenCount > 0){
            String l = sc.nextLine();
            if (l.matches(".*\\Q{\\E.*")) //Open brace
                curlyOpenCount++;

            if (l.matches(".*\\Q}\\E.*")) //Close brace
                curlyOpenCount--;

            ret += l + "\n";
        }

    }

    return ret;
}

public String MP_CHECKPOINT_OMP(String line, Scanner sc){
    return MP_CHECKPOINT(line, sc);
}

public String MP_CHECKPOINT_OMP_LOOP(String line, Scanner sc){
    return MP_CHECKPOINT_LOOP(line, sc);
}
```

```
public class Labels {
   Vector<Label> vLabels = new Vector<Label>();

   public void AddLine(String line){
      line = line.trim();
      String [] sect = line.split("\\{", 3); //Should have 3
         fields. (0)Pragma stuff, (1) Labels, (2) Variables

      //Pragma stuff is now nonsense to us. It just says this
         is a label/variable section.
      //So we get the other stuff:
      String [] lnames = sect[1].replace("{", "").replace("}",
         "").split("\\s+"); //Labels
      String tmp = sect[2].replace("{", "").replace("}", "");
      //Need to extract any in []s
      Vector<String> vvect = new Vector<String>();
      boolean inArr = false;
      String [] stmp = tmp.split("\\s+");
      String value = "";
      for (String s : stmp){
         if (s.startsWith("[")){
            inArr = true;
            value = s;
         }
         else if (s.endsWith("]")){
            inArr = false;
            value += " " + s;
            vvect.add(value);
            value = "";
         }
         else{
            if (inArr){
               value += " " + s;
            }
            else{
               vvect.add(s);
            }
         }
      }
      String [] vnames = vvect.toArray(new String
         [vvect.size()]);

      boolean found = false;
      for (String lname : lnames) {
         for (Label l : vLabels) {
            if (l.name.equals(lname)) {
               //Same label name. Thus, same label.
```

```
                found = true;
                //Add everything, then break.
                l.AddAllToLabel(vnames);
                break;
            }
        }

        if (!found) {
            //Add everything to a new label.
            Label nl = new Label(lname);
            nl.AddAllToLabel(vnames);
            vLabels.add(nl);
        }
    }

}

public class Label {
    String name;
    HashSet<PhoenixVar> vars = new HashSet<PhoenixVar>();
    boolean isOMP = false;
    boolean started = false; //Pre-metaprogramming scan will
        set this to true if start statement found.
    boolean firstStarted = false;

    public Label(String s){
        name = s;
    }

    public void AddAllToLabel(String [] vnames) {
        //Add everything, then break.
        for (String variable : vnames) {
            if (variable.matches("\\Q[\\E.*\\Q]\\E")) {
                //Array, form: [name size]
                String[] newArr = variable.replace("]",
                    "").replace("[", "").split("\\s+");

                if (newArr.length == 2) { //Array, no offset
                    AddVar(newArr[0], newArr[1], "0");
                }
                else if (newArr.length == 3) { //Array, offset
                    AddVar(newArr[0], newArr[1], newArr[2]);
                }
            }
            else { //Normal variable
                AddVar(variable, "-1", "-1");
            }
        }
```

```
}

public String MP_RestoreLabel(String ifstreamvar){
   String ret = null;

   //Dump everything into the file.
   String backupName = GetCstringName(basefilename, name);

   ret = "{\n"
         + "\tstd::ifstream " + ifstreamvar + ";\n"
         + MP_Declare_and_strcat_Backup(name, isOMP,
            basefilename)
         + "\t" + ifstreamvar + ".open(" + backupName +
            ");\n"
   ;

   for (PhoenixVar phoenixVar : vars){
      if (phoenixVar.nsize.equals("-1"))
         ret += "\t" + ifstreamvar + " >> " +
            phoenixVar.name + ";\n";
      else{
         String finish = phoenixVar.start + " + " +
            phoenixVar.nsize;
         ret += "\tfor(int i = " + phoenixVar.start + ";
            i < " + finish + "; i++)\n";
         ret += "\t\t" + ifstreamvar + " >> " +
            phoenixVar.name + "[i];\n";
      }
   }
   ret += "\t" + ifstreamvar + ".close();\n" +
      "}\n"
   ;

   return ret;
}

public String MP_BackupLabel(String ofstreamvar){
   String ret = null;

   //Read everything from the file.
   String backupName = GetCstringName(basefilename, name);

   ret = "{\n"
      + "\tstd::ofstream " + ofstreamvar + ";\n"
      + MP_Declare_and_strcat_Backup(name, isOMP,
         basefilename)
      + "\t" + ofstreamvar + ".open(" + backupName +
         ");\n"
```

```
            ;
            for (PhoenixVar phoenixVar : vars){
               if (phoenixVar.nsize.equals("-1"))
                  ret += "\t" + ofstreamvar + " << " +
                     phoenixVar.name + " << std::endl;\n";
               else{
                  String finish = phoenixVar.start + " + " +
                     phoenixVar.nsize;
                  ret += "\tfor(int i = " + phoenixVar.start + ";
                     i < " + finish + "; i++)\n";
                  ret += "\t\t" + ofstreamvar + " << " +
                     phoenixVar.name + "[i] << std::endl;\n";
               }
            }
            ret += "\t" + ofstreamvar + ".close();\n" +
                  "}\n"
            ;

            return ret;
         }

         public void AddVar(String n, String s, String s2){
            vars.add(new PhoenixVar(n, s, s2));
         }

         public class PhoenixVar {
            PhoenixVar(String n, String s, String s2){
               name = n;
               nsize = s;
               start = s2;
            }
            String name;
            String nsize; //Non-arrays default to size -1.
            String start; //Non-arrays default to offset -1.
         }
      }
   }
}
```

Bibliography

Duell, Jason (2005). "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart". In: *Berkeley, California. UNT Digital Library*. URL: http://crd-legacy.lbl.gov/~jcduell/papers/blcr.pdf.

Greg Bronevetsky Keshav Pingali, Paul Stodghill (2004). "Application-level Checkpointing for OpenMP Programs". In: *ACM SIGARCH Computer Architecture News* 32.5.07 - 13, pp. 235–247. URL: http://www.cs.cornell.edu/~stodghil/papers/sc05-openmp.pdf.

Jason Ansel Kapil Arya, Gene Cooperman (2007). "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: *Institute of Electrical and Electronics Engineers*. URL: http://www.ccs.neu.edu/home/kapil/papers/2009ipdps-dmtcp.pdf.

Yliluoma, Joel (2007). *Guide into OpenMP: Easy multithreading programming for C++*. URL: http://bisqwit.iki.fi/story/howto/openmp/.