Parallelization and the Application Programmer: Random Self-Oscillation or Old Faithful?

Celina Gibbs Yvonne Coady
University of Victoria
{celinag, ycoady}@cs.uvic.ca

Abstract

The typical application programmer is about to be swamped with a deluge of options for programming languages, libraries, frameworks and abstractions all designed to assist the inevitable need to parallelize new and existing codebases. Regardless of approach or level of abstraction provided, these programmers currently lack structured workflows to assist the development of parallel code.

Workflows within Software Development Life Cycle (SDLC) models, such as the Unified Process Model, have arguably structured modern practice. Though parallelism manifests itself as a seemingly innocuous non-functional requirement associated with performance, the ramifications of evolving an existing codebase often requires changes to mutiple workflows, spanning design, implementation, testing and maintenance.

In this paper we identify the need for an extension to traditional SDLC models to include systematic workflows structured around key causal relationships associated with parallelization. This feedback model, we call Geyser, is designed to link changes in design, platform, source, configuration, profiling and workload to corresponding impact on dynamic characteristics of the system. A rudimentary prototype tool demonstrates the possibility of leveraging this approach to infer the impact of change sets on performance.

Keywords parallel, workflow, software development lifecycle

1. Introduction

Transforming applications from sequential to parallelized versions requires changes that are rarely well contained within one application artefact (design documentation, build files, profiling infrastructure, OS configuration settings, source code, etc.). Instead, they causally cascade through multiple artefacts—a change in platform (hardware or OS) can force changes to source, causing the corresponding configuration or build to change. Subsequent changes in dynamic characteristics must be tracked, which may call for a change in profiling infrastructure, which may consequently impact memory footprint, adversely affecting performance, and so on.

Though current SDLC models support an iterative and structured approach to software development, they currently do not explicitly take into account the inherent complexities the dynamic consequences associated with parallel development. In order to get a clearer understanding of the specific artefacts involved, take for example the NAS Parallel Benchmarks (NPB) [10], which consists of eight programs designed to evaluate performance on parallel architectures. Though benchmarks in general are in a class of their own, collectively, these benchamarks represent essential parallel designs that cover a spectrum of computation and data movement characteristics that typically create resource contention and utilization challenges in parallel systems.

The results of running NPB 3.3 on a wide variety of platforms over several years are reported for over 25 different platforms [1]. Within each of these 200 sets of test results, the number of processors is varied, and a comparison is provided for *Class A* versus *Class B*, which differ in the size of their principal arrays. A sample of the benchmarks as run on a single processor are shown in Figure 1.

The developers of the benchmarks periodically release new results and further solicit additional results from the community at large. Each submission is required to include the following items in order to completely define a parallel application:

1. A detailed description of the **hardware and software configuration** used for the benchmark runs.
2. A description of the **implementation and algorithmic techniques** used.
3. **Source listings** of the benchmark code.
4. **Output listings** from the benchmarks.

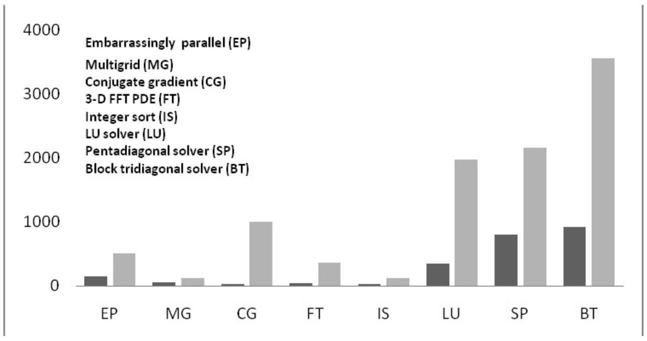


Figure 1. Comparison of Class A and Class B NPB 3.3 performance results on a single processor of a Cray Y-MP

As these benchmarks have withstood the test of time, we derive from this example the following list of explicit artefacts that are (sometimes subtly) causally related to the **dynamic characteristics** of a codebase: design documents, source files, configuration files, platform settings, profiling infrastructure and workload. This expansive list associated with parallelization is arguably substantially heavierweight and platform specfic than a typcial SDLC counterpart. We further propose that management of this lifecycle requires explicit efforts to structure workflows around the causal relationships between these concrete artefacts. In particular, we hope to assist developers reason about the ways in which changes that crosscut multiple artefacts contribute to corresponding changes in dynamic properties of the system, such as performance and resource utilization.

In the remainder of this paper we identify the need for an extension to traditional SDLC models to include systematic workflows structured around key causal relationships associated with artefacts that are subject to change during parallelization (Section 2). An investigation of three different parallel codebases uncovers representative examples of these artefacts and their subtle relationships (Section 3). Proposed integrated tool support designed to assist developers in tracking/infering changes during the evolution of a codebase from sequential to parallel is outlined (Section 4). Finally, a case study demonstrates prototype support for workflows that can be made more systematic and integrated across artefacts (Section 5).

2. Proposal

Workflows need to incorporate the ability to explore subtle feedback loops and causal relationships between changing artefacts in this intense development process. Figure 2 highlights these relationships in terms of changes, or Δs. For example, a Δ*platform* would mean a change to hardware or software infrastructure other than the source of a codebase (for example, an OS, or even runtime support for OpenMP). Δ*design* would be a change in the algorithm, accompanied by corresponding changes in the codebase. Not surprisingly,

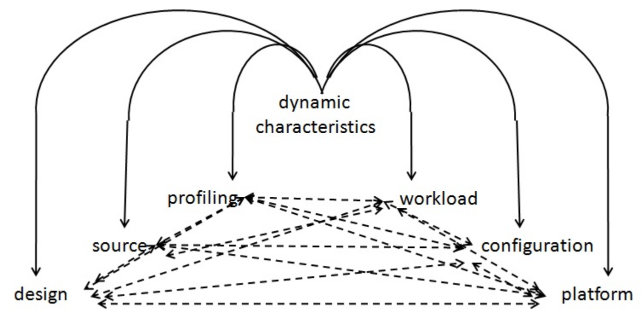


Figure 2. Geyser: Artefacts and Relationships

these coarse-grain tend to have correspondingly substantial impact on dynamic characteristics of a program.

To narrow the problem space, we focus on the following fine-grain artefacts and their causal impact on one or more dynamic characteristics (such as performance, memory usage, resource utilization and contention):

1. Δ*configuration*: A change to build or runtime options.
2. Δ*source*: A change to source of a codebase, including header files.
3. Δ*profiling*: A change in profiling infrastructure used specifically for dynamic assessment.
4. Δ*workload*: A change in workload/input.

Finally, completing the feedback loop:

Δ*dynamic*: A change in a dynamic characteristic that in turn may precipitate many more desired related changes to Δ*platform*, Δ*design*, Δ*configuration*, Δ*source*, Δ*workload*, and Δ*profile*.

The integration of feedback loops between sets of changes, $\Delta i \leftrightarrow \Delta j$, will most likely require system specific tools, but each of these tools can share common strategies. That is, the variability in everything from build systems to profiling strategies can be smoothed into a consistent development process to facilitate a common strategy for viewing and cataloging changes to sets of artifacts. Before proposing specific tool support for these workflows, we first survey several codebases to establish representative examples of the artefacts involved, and the relationships between sets of changes.

3. Investigating Artefacts

Managing and tracking cascading sets of changes associated with dynamic characteristics is difficult because the artefacts are often intertwined and change sets are the result of tight feedback loops. In order to establish best practices and tool support, we start by trying to gain a better understanding of the artefacts and relationships between changes.

3.1 Case Study: Harmony, NPB, OmpSRC

Here we consider examples from three codebases, the Harmony Portability Library [7], NPB [10], and OmpSRC [3], upon which we later derive a proposal for tool support.

3.1.1 The Harmony Portability Library

In terms of artefacts for configuration, each of the codebases we consider here leverage preprocessor directives heavily—a characteristic common to many C-based implementations. Previous work has demonstrated the many ways in which preprocessor directives have proven to be beneficial in implementing conditional code inclusion prior to a compilation phase [8, 9]. When multiple configurations of a system must be supported, conditional compilation is highly efficient in tuning a master code base for multiple versions of a program in which different performance, size, or functionality characteristics can be constructed. However, this fine-grained control over code placement associated with configuration and access to program state in the form of local variables and arguments makes it challenging for a developer to comprehensively manage source relevant to a specific build.

For example, a general design principle used in the Harmony Portability Library is to isolate platform-specific behaviour (e.g. opening a file or a socket) behind a well-defined, platform-agnostic, API boundary. In addition to providing greater flexibility in assembling software components, the API-based model keeps code readable, even at performance critical, low-level parts of the system, where macros are most commonly encountered. Figure 3 and Figure 4 show this in the case of redefining macros for semaphore support for configurations involving Linux and Windows, respectively.

```
/* SEM_CREATE */
#if defined(LINUX)
 #define SEM_CREATE(initValue)
  thread_malloc(NULL, sizeof(OSSEMAPHORE))
#else
 #define SEM_CREATE(initValue)
#endif
/* SEM_INIT */
#if defined(LINUX)
 #define SEM_INIT (sm, pshrd, inval)
  (sem_init((sem_t*)sm, pshrd, inval))
#else
 #define SEM_INIT(sm,pshrd,inval)
#endif
```

Figure 3. Macros in linux/thrdsup.h.

In the Linux build, the macros are defined either as containing functionality if the `LINUX` flag is defined, or empty otherwise (as indicated by `#else` in Figure 3). In the Windows build, the macros have only one definition (Figure 4). This means that, in the case of Linux, the developer not only must know which header (`.h`) files are included in the build, but also how the `LINUX` flag controls the implementation of these macros. It is thus critical for developers to be able to quickly determine the ramification of flag settings in terms of what code is included in a build.

```
/* SEM_CREATE */
/* Arbitrary maximum count */
#define SEM_CREATE(inval)
CreateSemaphore(NULL,inval,2028,NULL)
/* SEM_INIT */
#define SEM_INIT(sm,pshrd,inval)
          (sm != NULL) ? 0: -1
```

Figure 4. Macros in windows/thrdsup.h.

These examples demonstrate the intertwined nature of the relationships between changes to platform, changes to configuration, and changes to source that all impact dynamic characteristics of the system.

3.1.2 NAS Parallel Benchmarks

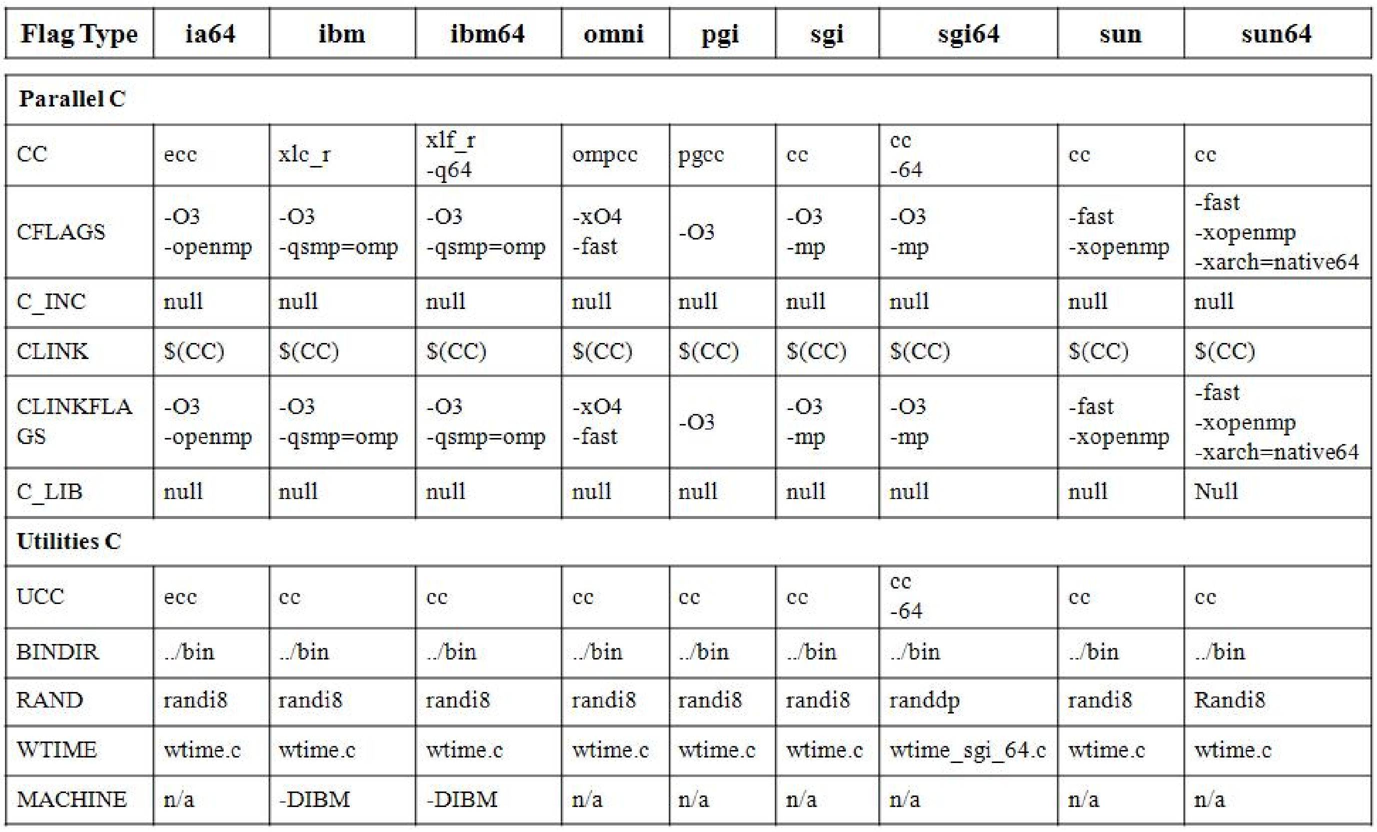
```
#ifdef _OPENMP
 adcpp->nTasks=omp_get_max_threads();
 fprintf(stdout,"\nNumber of available threads:
          %d\n", adcpp->nTasks);
 if (adcpp->nTasks > MAX_NUMBER_OF_TASKS) {
    adcpp->nTasks = MAX_NUMBER_OF_TASKS;
    fprintf(stdout,"Warning: Maximum number of
             tasks reached: %d\n", adcpp->nTasks);
 }
 #pragma omp parallel shared(pvstp) private(itsk)
#endif
 {
 double tm0=0;
 int itimer=0;
 ADC_VIEW_CNTL *adccntlp;
#ifdef _OPENMP
 itsk=omp_get_thread_num();
#endif
```

Figure 5. Example of combined pragma and flag usage in dc.c of NPBs.

```
#pragma omp parallel private(x,s,i,k)
   {
       INT_TYPE k1, k2;
       double an = a;
       int myid, num_procs;
       INT_TYPE mq;
#ifdef _OPENMP
       myid = omp_get_thread_num();
       num_procs = omp_get_num_threads();
#else
       myid = 0;
  num_procs = 1;
#endif
```

Figure 6. Example of combined pragma and flag usage in *is.c* of NPBs.

Table 1. Configuration file settings for OpenMP across nine architectures.



Flag Type	ia64	ibm	ibm64	omni	pgi	sgi	sgi64	sun	sun64
Parallel C									
CC	ecc	xlc_r	xlf_r -q64	ompcc	pgcc	cc	cc -64	cc	cc
CFLAGS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_INC	null	null	null	null	null	null	null	null	null
CLINK	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)
CLINKFLA GS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_LIB	null	null	null	null	null	null	null	null	Null
Utilities C									
UCC	ecc	cc	cc	cc	cc	cc	cc -64	cc	cc
BINDIR	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin
RAND	randi8	randi8	randi8	randi8	randi8	randi8	randdp	randi8	Randi8
WTIME	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime_sgi_64.c	wtime.c	wtime.c
MACHINE	n/a	-DIBM	-DIBM	n/a	n/a	n/a	n/a	n/a	n/a

The NPB have several configuration options designed to capture various aspects of sequential and parallel execution across nine different architectures. The support for this variety is characterized by the number of configuration artefacts (files) for the benchmarks. Each configuration file is based on a template *Makefile* for configuration options/flags associated with the build. For example, Table 1 demonstrates the fine grained nature of the flags and settings involved in configuring one instance of this set of benchmarks (OpenMP) across the architectures currently configured.

With respect to codebases that use OpenMP as a means to achieve parallelization, an additional artefact associated with a platform is the presence of absence of runtime support for the *pragmas* involved. As access to local variables is also involved, this typically is managed through combinations of conditional compilation for configuration and *pragmas*.

For example, the specific combination of conditional compilation and *pragmas* usage varies throughout NPB. Figures 5 and 6 illustrate the use of the `_OPENMP` flag to introduce OpenMP specific source. Figure 5 shows the OpenMP `#pragma` nested within a conditional `#ifdef _OPENMP`, that also contains `fprintf` statements for ad hoc profiling information, whereas Figure 6 shows the `#pragma` outside this configuration artefact and as part of the source—whether this runs or not now depends on the runtime platform.

Throughout NPB, several strategies for profiling are used. These include a mix of commented out print statements and conditional compilation based on the presence of an empty dummy file (`timer.flag`) added to the current working directory to turn on timing, creating profiling artefacts embedded in source as:

```
#ifdef  TIMING_ENABLED
    timer_stop( 2 );
    timer_start( 3 );
#endif
```

Additionally, emphasis is placed on profiling to provide configuration information during execution, augmenting output files with system characteristics. This comment is associated with the implementation of this profiling infrastructure within `sys\setparams.c`:

```
This is a gross hack to allow the benchmarks to print out
how they were compiled.
```

With respect to changes in workloads, input files for NPB are designed to vary the size of the input, and require slightly different configurations. For example, the `README` for OpenMP specifies that an increase in input size *may* require a larger stack size and gives the following suggestions for setting environment variables (configuration artefacts) for two specific architectures:

```
SGI Altix Intel compiler: setenv KMP_STACKSIZE 50m
SGI Origin3000: setenv MP_SLAVE_STACKSIZE 50000000
```

Collectively, these examples from NPB further demonstrate the subtle and intertwined nature of the relationships between changes to platform, profiling, configuration, source and workload that impact dynamic characteristics of the benchmarks.

3.1.3 OmpSCR

OpenMP Source Code Repository (OmpSCR) [3] is an infrastructure for benchmarking comprised of C, C++ and Fortran programs parallelized using OpenMP. These programs range from simple calculations to real scientific problems and are intended to both introduce OpenMP constructs and expose compiler weaknesses. These benchmarks provide further evidence that the manifestation of platform, configuration and profiling artefacts are consistent with those we found in Harmony and NPB.

For example, OpenMP uses a library routine to establish the maximum number of threads, shown in the following line from OmpSCR `c_md.c`:

```
NUMTHREADS = omp_get_max_threads();
```

which will use an environment variable, `OMP_THREAD_LIMIT`, to control the size of the thread population. Not surprisingly, there is a corresponding routine to set the number of threads OpenMP can create, as well as a *pragma* to locally set this number for a current parallel region, and an environment variable for establishing an initial value, `OMP_NUM_THREADS`. At any point in the source of an OpenMP program, the actual number of threads working depends on the order of execution order of a set routine versus the presence of a local *pragma*, or in the absence of both, the value of the environment variable.

This simple example further confirms the subtle nature of the dependencies between changes to platform, configuration and source that directly impact dynamic characteristics of these benchmarks.

3.2 Summary

Though all of these examples draw from low-level codebases, we believe these results generalize to other systems, applications and languages. Specifically, this case study serves to carve out the specific manifestations and relationships between the artefacts in the Geyser model, and the precise elements that need to be changed, tweaked, and tracked in the parallelization of codebases.

4. Assisting Workflows: Proposed Tools

Tools designed to assist workflows identified in Geyser must help track changes in dynamic properties of a system back to changes in artefacts that might be associated with that change. As a result, we need a consistent way of viewing changes and navigating between them. Figure 7 outlines what a customizable framework might look like, where each set of changes might be supported by a specific tool, but their integration is handled by a higher-level framework.

Unification of the views to a common format allows for a common starting point to iterate on feedback loops involving dynamic characteristics and series of causally related

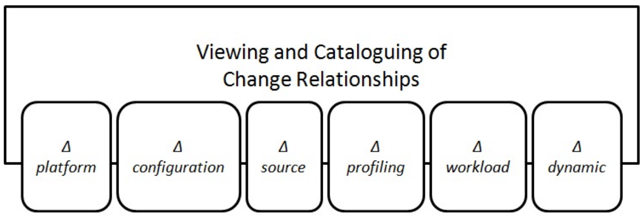


Figure 7. Structure of proposed framework with customizable extensions.

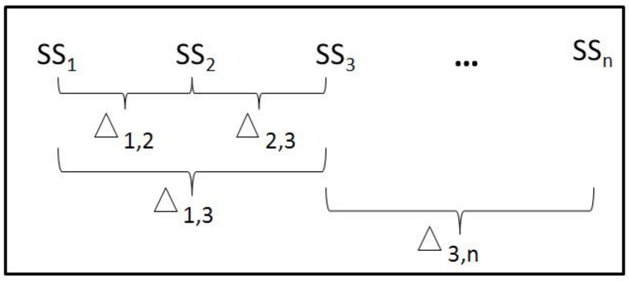


Figure 8. Snapshots (SS$_i$) of all system artifacts can be cataloged at any time, promoting further reasoning support for changes (Δ_i, j) between any two snapshots (SS$_i$, SS$_j$)

changes over time. Support for more global reasoning can be provided by saving system snapshots—current state for representations of each of the artifacts that collectively define a parallel application: design, platform, source, configuration, profiling, workload, and dynamic characteristics.

Though we envision policies to automate snapshot support, perhaps triggered by time or number of changes, our own prototype tools currently rely on manual practices to identify a semantically significant point in development, and a standard version control system for storage. Figure 8 highlights the ways in which the relative differences (Δi,j) between two or more snapshots (SSi, SSj) can be acquired.

The framework provides a language agnostic, semi-automated, general-purpose process for creating and visualizing performance results that are navigable to a parallel code base. This preliminary tool suite includes: (1) visualisation of static changes, (2) performance profiling of dynamic changes, and (3) an integration of static and dynamic views.

Visualisation of static changes we leverages the well established Eclipse plugin, C/C++ Development Toolkit (CDT) [2], which provides IDE support for C/C++ developers. CDT's integration with typical C/C++ tools such as g++ and gdb makes it an accepting base for this prototype tool suite. The `compare with` functionality of CDT supports comparison of text files within a workspace or across external folders and provides the support required for static change collection.

Performance profiling can be platform specific and can include a wide-range of options such as on-line and off-line modes and tracing capabilities. What is important to the tool suite is that the results, even including system resource utilization (core usage and memory), be made part of a system-snapshot in a consistent manner.

In our prototype tool we use the OpenMP specific profiling capabilities of ompP [5]. This tool uses automatic source code instrumentation of `#pragma`-marked parallel regions, collects profiling information at execution time and reports profiling results in detail. This mechanism specific profiling tool provides a detailed performance breakdown for each parallel region within an application. Each portion of code preceded by a `#pragma omp` is considered a region with ompP reporting the region type and location. Timing data and execution counts are provided on a per thread breakdown with a summation of all thread stats reported at the end.

Other profiling tools could easily be used within this tool suite, it is important to note that not all profilers take into account parallel execution. For example the GNU profiler gprof [6], flattens and sequentializes the otherwise parallel executable in the compilation process and the profiling is no longer useful for verification of non-functional requirements of parallel applications.

Integration of these tools should not only include support for navigation from performance results to static changes, but also a comprehensive means of tracking correlated static and dynamic changes over time, via system-snapshots.

Specifically, we have tailored our prototype to provide a visualisation of the profile results generated by ompP (performance and memory consumption). The prototype not only identifies areas of a code base that should be linked to runtime measurements and generates a visual representation of the performance differences between runs, but offers navigation to the static comparison support of CDT. The performance data is mapped to different versions of a system parsed by the source parser. This integration of views of static and dynamic changes allows for a common starting point to iterate on feedback loops involving dynamic characteristics and series of causally related changes over time.

5. Preliminary Case Study

As a preliminary evaluation of the Geyser model and associated tool support, we consider how a unified view of changes across different versions of a codebase might be presented, given the benchmark suite examples previously considered in Section 3.2.

5.1 Visualisation of Static Changes

A consistent way to view static changes in configuration files, source files and profiling that is typically embedded in source is from a high-level perspective, supported through *diffs* colour-coded in an Eclipse viewer. For example, Figure 9 provides a sample visualization of the three 64 bit architecture-specific configuration files for OpenMP benchmarks in coloured changes relative to the template configuration file provided. This view highlights the differences between the three configuration files and indicates that in each case (`ia64`, `ibm64`, `sgi64`), the changes to the template were made in roughly the same places.

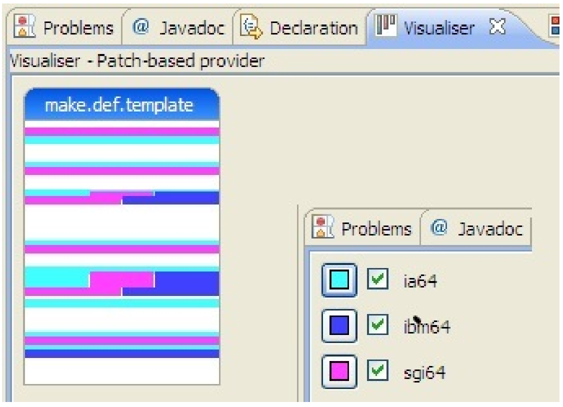
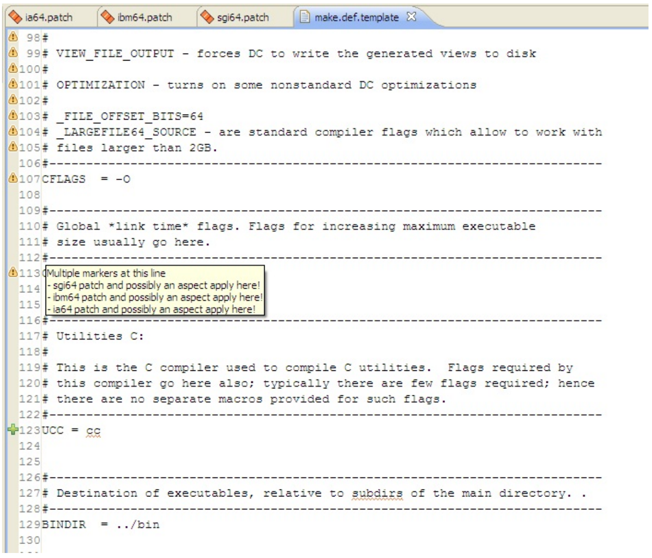


Figure 9. Visualisation of Δ*configuration* in NPB.

From such a view, Figure 10 demonstrates how navigation to the corresponding source view of these fine-grained differences. Gutter annotations indicate the places in which a specific configuration differs from the configuration template. The image shows the specific configuration differences of the `ia64` architecture, with the others tiled behind.



```
#
# VIEW_FILE_OUTPUT - forces DC to write the generated views to disk
#
# OPTIMIZATION - turns on some nonstandard DC optimizations
#
# _FILE_OFFSET_BITS=64
# _LARGEFILE64_SOURCE - are standard compiler flags which allow to work with
# files larger than 2GB.
#---------------------------------------------------------------------------
CFLAGS   = -O

#---------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#---------------------------------------------------------------------------
Multiple markers at this line
- sgi64 patch and possibly an aspect apply here!
- ibm64 patch and possibly an aspect apply here!
- ia64 patch and possibly an aspect apply here!
#---------------------------------------------------------------------------
# Utilities C:
#
# This is the C compiler used to compile C utilities.  Flags required by
# this compiler go here also; typically there are few flags required; hence
# there are no separate macros provided for such flags.
#---------------------------------------------------------------------------
UCC = cc


#---------------------------------------------------------------------------
# Destination of executables, relative to subdirs of the main directory. .
#---------------------------------------------------------------------------
BINDIR   = ../bin

```

Figure 10. Fine-grained flag configuration view for ia64.

In this figure, simple flags control the different options for each architecture. This navigation facilitates a developer's ability to experiment with different combinations of flag settings. The importance of this type of experimentation is alluded to in the OpenMP README file which states: `For some buggy OpenMP compilers, you may have to play with the optimization flag, for instance, use "-O2" over "-O3".`

Similarly, Figure 11 provides a high-level perspective of where parallelization is introduced by both OpenMP (dark) and MPI (light) with respect to the serial implementation. This view shows the larger impact of the MPI implementation as well as the overlap between the two implementations. As demonstrated in Figure 10, direct navigation to the source for the examination of implementation details provides an integrated view of changes across versions.

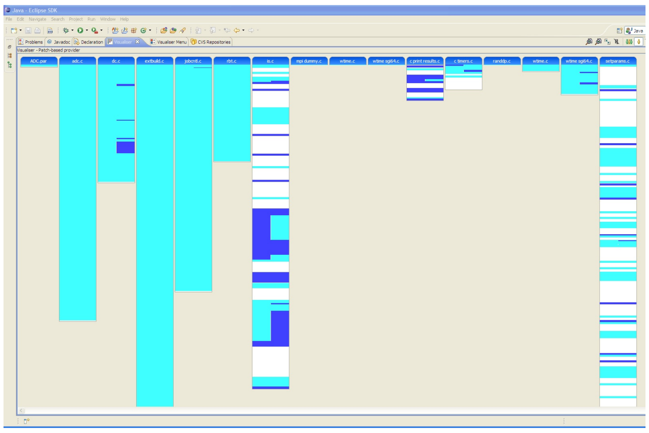


Figure 11. Visualisation of Δ*source* in NPB.

Finally, Figure 12 provides the visualization of a naive implementation of profiling which builds on existing profiling support provided by the `TIMING_ENABLED` flag within the NPB-IS benchmark. The implementation introduces two levels of profiling: `PROFILE_ENTER` and `PROFILE_LEAVE`, where the first traces entry to a function and a combination traces function entry and exit. The visualizer integrates a high-level view of profiling as it applies to the base system, as well as the ability to configure the profiling through navigation to the editor view.

5.2 Integrating Views of Static and Dynamic Changes

To demonstrate the integration of static and dynamic deltas we provide a proof of concept example, leveraging our previous work with tools applied to the parallel domain. In this example we focus on the molecular dynamic (`c_md.c`) from the OmpSCR benchmark suite scientific application which has two parallelizeable regions.

Our previous work with tools [11] focused on support for reasoning about conditionally compiled source code based on a set of user specified conditional flag settings. This initial work provided source code views with dead code either grayed out or folded out of view within the Eclipse [4] IDE viewer. This functionality supports the ability to experimentally and iteratively refine configuration setting through changes in flag assignments.

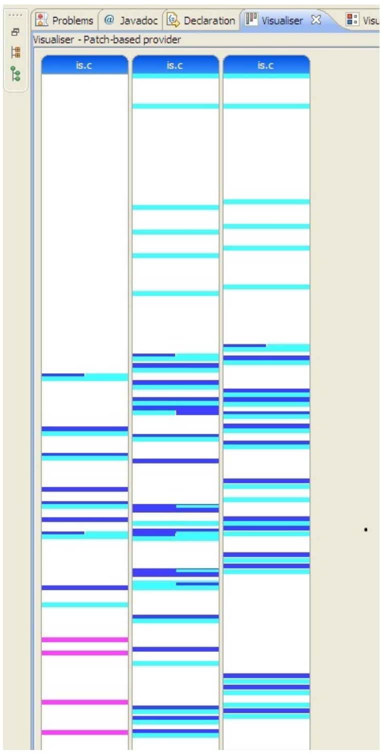
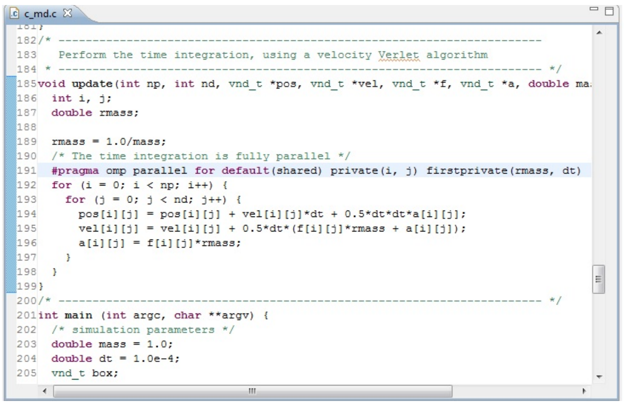


Figure 12. Visualisation of Δ*profiling* in NBP.

By merging the source visualisation capabilities of Eclipse with performance capturing capabilities of the OpenMP Profiler, ompP [5] for collection of dynamic data, we further support workflows according to the Geyser model. This integration is facilitated by an automatic parsing of both the source code and its corresponding the performance data. A graph of the ompP output, as seen in Figure 14, provides a visualization of the dynamic differences between multiple configurations. Each bar in the graph represents the omPP output of a single configuration, corresponding to the Eclipse project the source code resides in.

In this example, a C application whose source is a single file, `c_md.c`, has four different versions. Each version with a region that supports parallelization defined by *pragmas*. Each of the four versions are contained in separate C projects in Eclipse. Clicking on one of the bars in this chart will open the corresponding version of `c_md.c` within the C editor and highlight the first parallel region as identified by `#pragma` parser. This Eclipse view, shown in Figure 13 allows a user to reason about the which region-changes may be responsible for a causal change in performance. In this figure the two versions of the source code that correspond to the `FullParallel` and `TimeParallel` performance entries in the bar chart are shown tiled within the Eclipse view.

Developers need to adopt structured and systematic workflows to analyse how different versions of parallel source code impact performance of an application. Figure 14 demonstrates a navigable view of four different versions of the



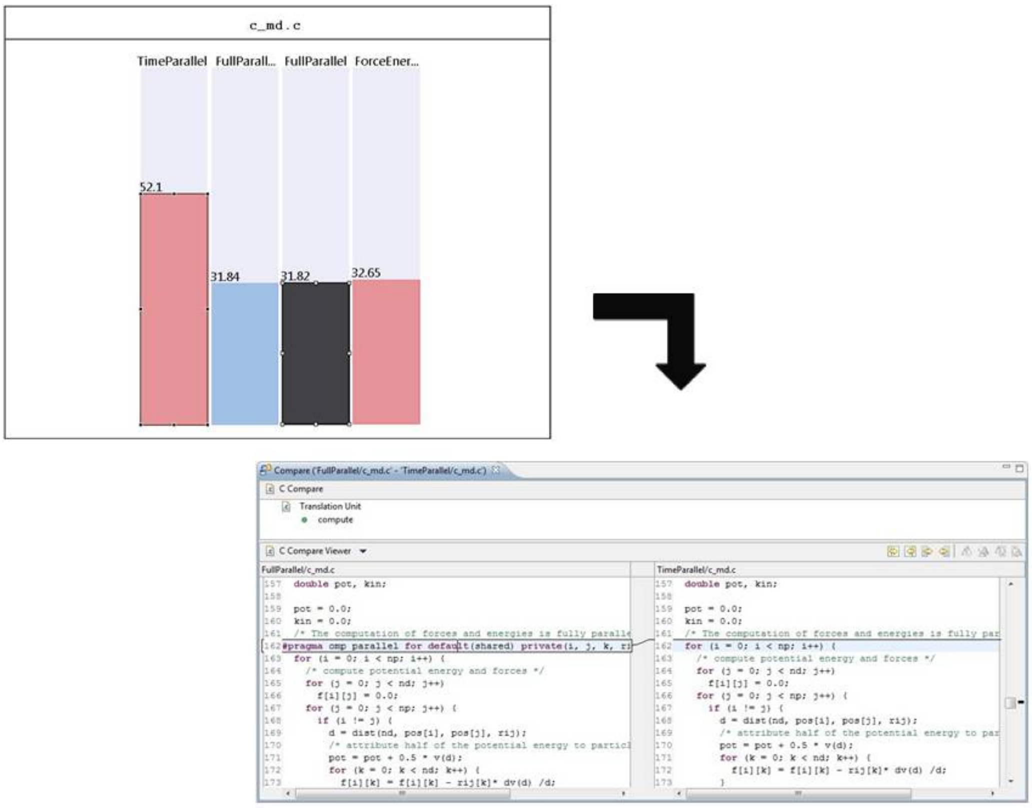
```
/* ---------------------------------------------------------------------
   Perform the time integration, using a velocity Verlet algorithm
 * --------------------------------------------------------------------- */
void update(int np, int nd, vnd_t *pos, vnd_t *vel, vnd_t *f, vnd_t *a, double ma
  int i, j;
  double rmass;

  rmass = 1.0/mass;
  /* The time integration is fully parallel */
  #pragma omp parallel for default(shared) private(i, j) firstprivate(rmass, dt)
  for (i = 0; i < np; i++) {
    for (j = 0; j < nd; j++) {
      pos[i][j] = pos[i][j] + vel[i][j]*dt + 0.5*dt*dt*a[i][j];
      vel[i][j] = vel[i][j] + 0.5*dt*(f[i][j]*rmass + a[i][j]);
      a[i][j] = f[i][j]*rmass;
    }
  }
}
/* --------------------------------------------------------------------- */
int main (int argc, char **argv) {
  /* simulation parameters */
  double mass = 1.0;
  double dt = 1.0e-4;
  vnd_t box;
```

Figure 13. Pragma view within the Eclipse editor.

molecular dynamic application with the OmpSCR benchmark. By selecting two or more configurations, that is clicking on two bars in the performance graph, the corresponding source files can be compared. In this case, the two different versions of `c_md.c` as seen in the lower portion of Figure 14.

The differences between the two source versions, in this simple case, just an OpenMP `#pragma` on line 162 of the left panel, can be highlighted. This allows the user to directly navigate from performance graph to the source, and view the source changes that may have contributed to the performance differences seen in the graph.

In this example, it is evident that the `FullParallel` versions result in better performance as the two `FullParallel` versions have shorter bars than the versions with partial parallelization. The bar length, the colouring of the selected base bar, and the colour coding of the remaining bars aide in this visual configuration comparison.



FullParallel/c_md.c

```
  double pot, kin;

  pot = 0.0;
  kin = 0.0;
  /* The computation of forces and energies is fully paralle
#pragma omp parallel for default(shared) private(i, j, k, ri
  for (i = 0; i < np; i++) {
    /* compute potential energy and forces */
    for (j = 0; j < nd; j++)
      f[i][j] = 0.0;
    for (j = 0; j < np; j++) {
      if (i != j) {
        d = dist(nd, pos[i], pos[j], rij);
        /* attribute half of the potential energy to particl
        pot = pot + 0.5 * v(d);
        for (k = 0; k < nd; k++) {
          f[i][k] = f[i][k] - rij[k]* dv(d) /d;
```

TimeParallel/c_md.c

```
  double pot, kin;

  pot = 0.0;
  kin = 0.0;
  /* The computation of forces and energies is fully par
  for (i = 0; i < np; i++) {
    /* compute potential energy and forces */
    for (j = 0; j < nd; j++)
      f[i][j] = 0.0;
    for (j = 0; j < np; j++) {
      if (i != j) {
        d = dist(nd, pos[i], pos[j], rij);
        /* attribute half of the potential energy to par
        pot = pot + 0.5 * v(d);
        for (k = 0; k < nd; k++) {
          f[i][k] = f[i][k] - rij[k]* dv(d) /d;
        }
```

Figure 14. Navigation with Eclipse.

6. Conclusion

Returning to our original question regarding *random oscillation* verses *old faithful*, we believe systematic software development practices need to be defined to guide the application programmer who must now wrestle with the tasks involved with parallelization. Inevitably, feedback with dynamic characteristics must be included in workflows and systematically explored, regardless of language or abstractions used. In that regard, based on our assessment of parallel codebases, artefacts involved, and their relationships, we hope the Geyser model can provide valuable assistance to application developers.

Once we can agree upon the workflows associated with parallelization, we will be able to begin to more fully define a consistent framework for tool support. Currently, we have developed a prototype based on the Geyser model, providing a rudimentary integration of static and dynamic views. We believe this will be necessary support for evolving sequential codebases to efficient incarnations of their parallel counterparts.

References

[1] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-2630-5.

[2] E. CDT. CDT Project. http://www.eclipse.org/cdt/.

[3] A. J. Dorta, C. Rodriguez, and F. de Sande. The openmp source code repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244–250, 2005. doi: 10.1109/EMPDP.2005.41. URL `http://dx.doi.org/10.1109/EMPDP.2005.41`.

[4] Eclipse. The Eclipse Framework. http://www.eclipse.org/, 2008.

[5] K. Furlinger, M. Gerndt, and T. U. Munchen. M.: ompp: A profiling tool for openmp. In *In: Proceedings of the First International Workshop on OpenMP (IWOMP 2005*, 2005.

[6] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989401.

[7] Harmony. Harmony Home Page. `http://harmony.apache.org/`.

[8] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. C/c++ conditional compilation analysis using symbolic execution. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 196, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0.

[9] G. J. B. Michael D. Ernst and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions Software Engineering*, 28(12):1146–1170, November 2002.

[10] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[11] N. Singh, C. Gibbs, and Y. Coady. C-CLR: a tool for navigating highly configurable system software. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software (ACP4IS)*, page 9, New York, NY, USA, 2007. ACM. ISBN 1-59593-657-8. doi: http://doi.acm.org/10.1145/1233901.1233910.