DOI: 10.14529/jsfi150401

Live Programming in Scientific Simulation

Ben Swift[1]***, Andrew Sorensen***[1]***, Henry Gardner***[1]***, Peter Davis***[1]***,***
Viktor K. Decyk[2]

We demonstrate that a live-programming environment can be used to harness and add run-time interactivity to scientific simulation codes. Through a set of examples using a Particle-In-Cell (PIC) simulation framework we show how the real-time, human-in-the-loop interactivity of live-programming can be incorporated into traditional "offline" and development workflows. We discuss how live programming tools and techniques can be productively integrated into the existing HPC landscape to increase productivity and enhance exploration and discovery.

Keywords: live programming, particle-in-cell, JIT-compilation.

Developing simulation codes which effectively model scientific phenomena, enable exploration and discovery, and run efficiently on modern heterogeneous HPC architectures is a difficult and demanding undertaking. One of the reasons for this is a very slow feedback loop between code development, execution and analysis. In this paper, we show how tools and techniques from live programming can constitute a dramatic intervention in HPC software development, modification, tuning and deployment. Using the Extempore [15] live-programming environment we demonstrate how procedure-level "hot-swapping" (the ability to change an executing procedure while it is running) can be added to existing simulation codes (written in highly optimised languages such as C/C++/Fortran) in order to provide HPC application developers with the level of interactivity which is now the characteristic of software development in other modern application domains.

This paper presents a case study that starts with a mature Particle-In-Cell (PIC) simulation framework that has been designed for next-generation, heterogeneous HPC architectures [3]. In a tutorial-like fashion, we show how this code can be harnessed and run interactively using the Extempore live-programming environment. This harnessed code then allows the programmer to interact with and modify it in ways unenvisioned by the original developers. Following our presentation of this case study, we then discuss the wider prospects for incorporating this type of harnessing and live-programming in the development and deployment of HPC scientific simulation codes.

1. Outline of a PIC simulation code

The Particle-in-Cell (PIC) technique [2] is widely used in plasma physics and plasma engineering. In the technique, plasmas are modelled by tracking the trajectories of many particles interacting self-consistently with external electric and magnetic fields. Various domain decomposition techniques (e.g. [9]) have been developed for running these codes efficiently across parallel architectures and this area of scientific simulation is a heavy user of HPC. The specific PIC codes we use in this paper are "skeleton" PIC codes written in C and Fortran [3]. These codes have been developed for exploring new computational architectures [3] and as a foundation for teaching and reasoning about PIC simulation. They support various parallelisation schemes, including shared memory (OpenMP), distributed memory (MPI) and GPU accelerators (CUDA).

[1]Australian National University, Canberra, Australia
[2]University of California, Los Angeles, USA
[3]`http://picksc.idre.ucla.edu/software/skeleton-code`

Although there are (sometimes subtle) variations between specific PIC approaches, the main simulation loop has three steps [3].

1. **deposit** the electric charge (or similar) from the particles into a grid
2. **solve** the field equations to obtain the fields (electric or electromagnetic)
3. **push** (move) the particles in response to that field

The data structures which are updated during this loop are the particle coordinates and the fields.

The basic structure of a representative PIC skeleton code (in C but with details omitted for clarity) is shown in Figure 1. The subroutines `deposit`, `solve` and `push` do the main work of

```
int num_particles = 1e6;
int grid_size = 512;

/* initialise the particle and field data */
float* part = init_particles(num_particles);
float complex* field = init_fields(grid_size);

/* main simulation loop */
while (step < num_steps)
{
  deposit(part);                /* deposit charge on grid */
  solve(part, field);           /* calculate field */
  push(part, field);            /* move particles */
  step++;                       /* increment timestep */
}
```

Figure 1. Basic PIC code structure (in C with details omitted for clarity)

the simulation. Each of these procedures can be quite sophisticated; they may run in parallel across a distributed-memory compute cluster using MPI and they may also take advantage of GPU acceleration.

In the conventional workflow, a PIC simulation code is compiled and linked into an executable which is then run either on a single machine or across a cluster (using `mpirun`). In the next section we will show how the Extempore live-programming environment can be used to harness and run such a simulation in an interactive manner.

2. Real-time intervention in PIC simulation

In order to make our C-language PIC code interactive, it needs to be compiled to generate a *shared library* (e.g. with GCC's `-shared` flag). We can then start up the Extempore live coding environment as a standalone process on the host computer or on a cluster through `mpirun` or some other remote execution mechanism. The Extempore instance binds a TCP socket and waits, listening for code to execute. The programmer can then load and bind the PIC routines and variables from the shared library into a running Extempore process.

Figure 2 shows the main PIC simulation loop, equivalent to Figure 1, as it would be run from Extempore. The Extempore language uses an s-expression syntax which we will not describe in detail here. The feature of this figure is that the simulation routines `deposit`, `solve` and `push`

directly address machine code generated from the original C code, but the toplevel `while` loop is now written in Extempore. The primary simulation loop is now controlled from Extempore, even though the heavy computation is still carried out by procedures compiled from C.

```
;; this is just a simple 1D pic code, for clarity
(bind-val num_particles i32 1e6)
(bind-val grid_size i32 512)

;; initialise the particle and field data
(bind-val part float* (init_particles num_particles))
(bind-val field float* (init_fields grid_size))

;; main simulation loop
(while (< step num_steps)
  (deposit part)                ;; deposit charge on grid
  (solve part field)            ;; calculate field
  (push part field)             ;; move particles
  (set! step (+ step 1)))       ;; increment timestep
```

Figure 2. PIC code structure in Extempore equivalent to Figure 1

Once it is harnessed in Extempore, the programmer can now *interactively* send one or more s-expressions (Extempore "statements") to the Extempore compiler. These expressions are then "just-in-time" (JIT) compiled into native machine code through an LLVM [8] back-end and begin executing immediately. At this point, the execution of the program will produce the same results as the C code in Figure 1 (including any MPI-based communication) so long as the same procedures are being called in their correct sequence.

Once the PIC simulation code is running in Extempore, the programmer is able to make changes to the source code. Once these changes are complete, re-evaluating the relevant code will re-compile that code chunk (e.g. a subroutine) and hotswap it into the next loop of the running program. For example, we might add a call to some (pre-prepared) visualisation subroutines into our main loop, as shown in Figure 3. In the next iteration of the main loop, the visualisations would be drawn and then updated for each step through the simulation loop. The rest of the computation, including the state of the particles and the fields, will be preserved unchanged. A screenshot from a PIC simulation corresponding to the code in Figure 3 is shown in Figure 4, and it can also be seen "in action" in a video accompanying this paper.[4]

Extempore is designed to mix the high-level expressiveness of Lisp with the low-level expressiveness of C. Extempore brings modern language features, such as an advanced type system, type inference, strong temporal semantics, reified generics, first class closures and macros, together with low-level expressivity, including direct pointer manipulation, explicit memory management, architecture width primitives and strong C-ABI compatibility. Because of these design considerations, and despite the dynamic interactivity of the live-programming workflow in this example, the performance of the main loop is comparable to that of the original C code (see Table 1). This efficiency is in marked contrast to other dynamic "glue-language" approaches

[4] `https://vimeo.com/126577281`

```
;; main simulation loop
(while (< step num_steps)
  (deposit part)
  (solve part field)
  (push part field)
  ;; add the visualisation routines
  (draw_particles part)
  (draw_field field)
  (set! step (+ step 1)))
```

Figure 3. Adding a call to the visualisation routines into the running main loop

```
(bind-func kick_particles
  (λ (np vx vy)
    (doloop (i np)
      (pset! part
             (+ (* i 5) 2)
             (+ vx (pref part (+ (* i 5)
2))))
      (pset! part
             (+ (* i 5) 3)
             (+ vy (pref part (+ (* i 5)
3)))))))

(bind-func external_field
  (λ ()
    (let ((nfield
           (* nxe (pref nyp_ptr 0))))
      (doloop (i nfield)
        (pset! (cast bxyz float*)
               (* i 3) .0))))
```

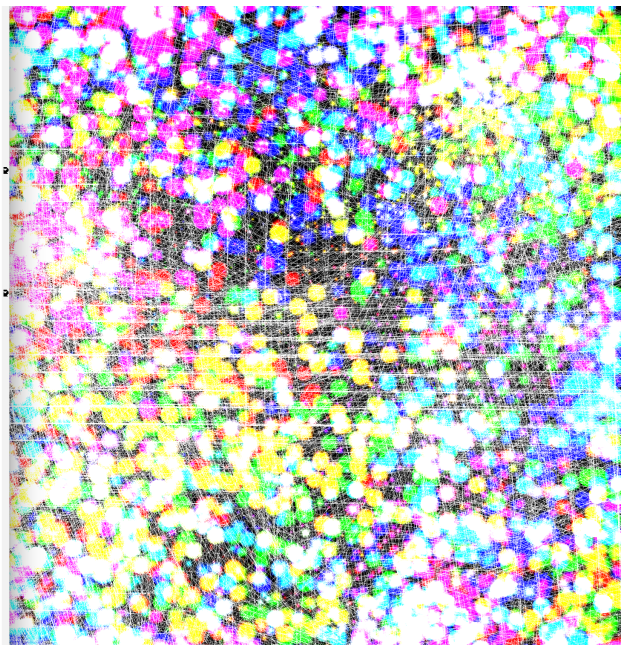

Figure 4. A screenshot of the running PIC simulation with a live visualisation of the particles. Here the visuals are shown next to the Extempore code in the text editor, but they could be shown on any monitors (including being streamed over the network)

(e.g. NumPy [13]) which exhibit good performance when the execution stays within the underlying compiled Fortran libraries, but poor performance if it "bubbles up" into the python wrapper code[5]. In live programming it is natural and, indeed, desirable to have such a "bubble up" in order to enable free and fluid steering, tuning, diagnosis and discovery in scientific simulation. Accordingly, live-programming environments for HPC need to generate fast executable code, whilst also providing programmers with more expressive tools to better manage the time and space demands of these complex, running simulations.

As a demonstration of the low overheads in using Extempore to harness a PIC simulation in a parallel environment, we have performed a number of benchmarking runs on a small OpenStack-based cloud cluster (4 × Intel Xeon compute nodes, 8 cores & 64GB ram per node). Our code uses MPI for distributed-memory processing, and contains a mixture of Extempore code and calling pre-compiled C subroutines as described above. The results are shown in Table 1. From

[5]For an excellent analysis of the "bubble up" performance problems associated with the R statistics language see [11]

the table, it is clear that the Extempore overheads are largely confined to a start-up cost of 20 seconds and thereafter the Extempore version has almost identical performance to the C code on up to 32 cores and 1.44×10^8 particles.

Table 1. Comparison of the PIC code running time as a compiled C program vs running live in Extempore, calling subroutines from a shared library. Total wall clock time is shown, with average and standard deviation calculated over three runs

environment	cores	particles	time (sec)	time s.d.
C	8	1.44×10^8	1574	1.38
	16	1.44×10^8	834	1.88
	32	1.44×10^8	482	1.35
Extempore	8	1.44×10^8	1608	1.3
	16	1.44×10^8	858	0.67
	32	1.44×10^8	496	0.77

As a further extension of our case study, we will now intervene in our running simulation to add an external electric field. This is shown in Figure 5 where we add an inner loop to the main simulation which manipulates data in the `field` region of the heap to add an external electric field which sinusoidally varies over the domain. When the main loop is re-evaluated after this change, the programmer can immediately see the way this field affects the particle behaviour through the real-time visualisation added in the previous step. This instant feedback is extremely useful in parameter tuning, where different values can be tried until a desired result is achieved.

```
;; main simulation loop
(while (< step num_steps)
  (deposit part)
  (solve part field)
  (push part field)
  (draw_particles part)
  (draw_field field)
  ;; add an external electric field
  (doloop (i grid_size)
    (+= field (cos (* 2PI (/ i grid_size)))))
  (set! step (+ step 1)))
```

Figure 5. Adding an external electric field to the simulation by directly writing a new tight loop into the main simulation loop

As a final example, we can allow more of our original PIC simulation code to "bubble-up" into the Extempore layer by re-writing one of the main simulation routines. Figure 6 shows a `deposit` subroutine in Extempore code, which has the same behaviour as the `deposit` subroutine in the original C code. In the "deposit charge" step of the PIC simulation, the charge density is estimated by calculating the contribution of each particle to nearby grid points. In this first-

order code, each particle's contribution to the total charge density is considered only at the closest two grid points (i.e. above and below the particle's position).

```
(bind-func deposit
  "for each particle, deposit the charge on grid using
1D linear interpolation"
  (lambda (part)
    (doloop (j num_particles)
      (let ((x (pref part j))  ;; read particle x position
            (x_grid (floor x))   ;; xpos of lower gridpoint
            (idx (convert x_grid i32))
            (dx (- x x_grid)))   ;; distance to lower gridpoint
        ;; interpolate & deposit charge at lower grid point
        (+= q idx (* charge_per_particle x_grid))
        ;; interpolate & deposit charge at upper grid point
        (+= q (+ idx 1) (* charge_per_particle (- 1.0 dx)))))))
```

Figure 6. The "deposit charge on grid" subroutine

Replacing subroutines from the original C code with new Extempore variants provides increasing levels of run-time interactivity, as each new Extempore routine can now be further modified and refined on-the-fly. Figure 7 shows how the new Extempore `deposit` subroutine can be updated to use cubic interpolation. This gives increased simulation accuracy at the expense of increased computational complexity. Again, since the visualisation routines are still present in the main loop, the programmer can see the results of this new `deposit` subroutine when it is hot-swapped into the main loop. No recompilation of the main loop is required — the new `deposit` routine will be called automatically on the next iteration of the loop.

This last part of our example scenario is complicated enough that we would need to have a good reason to write it live for a production code. However, the live approach might speed up the discovery process, particularly at the prototyping phases of software development. Some low-order interpolation schemes can be unstable, sometimes violently and dramatically. Comparing different solvers or boundary conditions in a live harness could speed up the feedback loop and be very insightful.

```
(bind-func deposit
    "same as before, but with *quadradtic* interpolation"
  (lambda (part)
    (doloop (j num_particles)
      (let ((x (pref part j))  ;; read particle x position
            (x_grid (floor (+ x .5)))   ;; xpos of lower gridpoint
            (idx (convert x_grid i32))
            (dx (- x x_grid)))   ;; distance to lower gridpoint
        (+= q idx       (* .5 charge_per_particle (pow (- .5 dx) 2.)))
        (+= q (+ idx 1) (* charge_per_particle (- .75 (pow dx 2.))))
        (+= q (+ idx 2) (* .5 charge_per_particle (pow (+ .5 dx) 2.)))))))
```

Figure 7. An updated "deposit charge on grid" subroutine which uses quadratic interpolation in the charge depositing step, so that the charge from each particle is deposited to the *three* closest grid points. This is more accurate than the original linear `deposit` routine shown in Figure 6, but also more computationally expensive

2.1. Prospects for Discovery in PIC Simulation

The above examples constitute the first effective demonstration of live-programming harnessing of legacy scientific software in the literature that we are aware of (certainly this is the case for Particle in Cell simulation). Although they can be thought of as being illustrative, the demonstrated effectiveness of these examples paves the way for imagining future prospects for discovery.

One HPC scenario that comes to mind is that of a harnessed PIC simulation of an "always on" plasma like a steady-state plasma experiment or a steady-state model of an astrophysical system such as the sun. Within a live-coding harness one could easily undertake live numerical experiments for discovering new insights into plasma behavior. An important tool for such discoveries is the frequency spectrum, sometimes called spectroscopy in other fields. To compute this numerically we would take data (the plasma potential or the electric or magnetic fields) from the last N time steps, e.g., N=1000. In plasmas these physical quantities oscillate collectively at specific frequencies. In the simulation, we would compute a Fourier time integral such as $\int f(k,t)exp(-i\omega t)dt$, where t varies for these 1000 time steps, for each value of k. We would display this as a function of ω versus k, such as a color map or contour plot. When the plasma advances one step, we would throw away the oldest data point and add the newest one. Thus we would have a snapshot of the current frequencies for various wavelengths as the simulation proceeds. This could reveal many non-linearities that might generate new frequencies, or amplify existing frequencies in the spectrum. For example, the insertion of a substantial group of particles moving with a velocity $\mathrm{v_{beam}}$ could lead to the growth of waves whose frequency is given by $\omega = k\mathrm{v_{beam}}$. As another example, in the PIC codes by default the ions are a neutralizing background. This could be modified to give them a sinusoidal density perturbation (which can be fixed, like the fixed electric field mentioned above.) The particles moving across such an ion profile might generate new plasma frequencies via a parametric process. On the other hand, some of these new waves might also be "fake physics" coming from numerical errors in the simulation. Through a trial and error process of discovery it is possible that new and interesting plasma phenomena could be revealed and distinguished from numerical artefacts.

3. Considering Liveness

Up until now we have only briefly discussed potential benefits of liveness in scientific software development and simulation. In this section we highlight a few specific areas where we believe that the liveness will become important.

3.1. Steering

Computational steering [12] has long promised benefits for HPC and computational science. In 2007, the National Science Foundation (NSF) workshop [5] highlighted the need for dynamic interactivity as a looming "grand challenge" in scientific computation and data analysis. In particular, the view of the workshop was that "systems need to become more dynamic and amenable to steering by users and be more responsive to changes in the environment" [5, p31]. In a more recent survey of the field, Mattoso et al. [10] discussed the current landscape in regard to dynamic HPC workflow steering. They found that while progress has been made in some aspects, such as the ability to cancel or re-distribute HPC subtasks and the ability to modify particular parameters and convergence conditions at run-time, domain-level dynamic

interactivity, such as inserting low-level debugging statements into running code, or adding new behaviours to a running simulation, remains an open challenge. It is this challenge that we believe can be met through the incorporation of live-programming tools and techniques into dynamic HPC workflows. Our example of the always-on plasma simulation in the previous section is one possible steered HPC workflow.

3.2. Rapid Prototyping in Software Development

Just as "discovery" in HPC is a slow and painful process, HPC architectures themselves are notoriously difficult to program, and their problem domains are often not well understood from a computational standpoint. A significant risk for HPC programs is that of over-investing substantial resources into large "waterfall" development projects. One risk mitigation strategy is to focus heavily on early prototyping including repeated execution of code versions which may even fail being incorrect ("fail fast, fail often" is one of the new software development mantras). While the cavalier nature of this process may strike some as profoundly unsuited to the rigorous demands of good engineering, the value of early rapid prototyping is now widely regarded as best practice even in the HPC community [7].

There are, of course, costs associated with this early prototyping. Prototypes are often developed on a single machine, running serial codes, written in a higher level language such as Python. It can become difficult to later disentangle the high-performance production code from these higher-level languages.

In contrast, the live-programming approach described here can enable prototyping to happen on small development clusters, running parallel algorithms with a high performance language from the very outset. The interactive, programmer-driven *evaluate-compile-execute* workflow works even in a distributed-memory cluster environment (see Table 1). Extempore processes are network addressable, and can be run in a distributed fashion across a cluster of nodes. Code can be sent (over a TCP connection) from a developer's workstation for evaluation on either one, some, or all of the nodes in the cluster. In this way it is even possible to have different programs running on each node which could be especially useful for debugging situations where the data and behaviour of a single node can be explored and contrasted with other live simulations on other nodes. Using the same language for prototyping and production saves considerable re-engineering down the track.

We believe that live programming environments like Extempore provide a key advantage by allowing developers to move seamlessly from prototyping to production. Indeed, under ideal circumstances it should be difficult to identify when a project moves from prototyping into production, and then back to prototyping in order to properly support the ongoing development and maintenance of the system. In short, the distinction between prototyping and production is lost as live-programming integration times become increasingly short.

3.3. Resource Management

System resource management is becoming an increasingly disparate interplay of heterogenous hardware running heterogenous systems residing in a heterogenous cloud. The great complexity that this picture paints is also a source of great opportunity to develop solutions that can be more easily tailored to target the exact requirements of the problem at hand. Such solu-

tions would provide the potential for huge cost savings, and a more equitable playing field for large-scale "on-demand" scientific computing.

Taking advantage of these opportunities places ever greater demands on software developers, who must begin to integrate many aspects of resource management, which were previously external concerns, into their software projects. In a real sense HPC site managers and software developers are more exposed to the physical hardware now than they have been for a generation including consideration of aspects, such as processor affinity, memory locality, vector register sizes, phi-cores and GPGPU, cachelines, node management, code scheduling, network synchronization and time management.

We believe that live programming can help to manage these complexities by making resource management solutions more dynamic. The ability to design and control resource management solutions directly in a live programming environment presents opportunities for monitoring resource usage more effectively and adapting more rapidly to changing requirements. To take advantage of these opportunities the roles of developers and technical operations have become increasingly blurred, which has in turn lead to a growing interest in "DevOps" — the integration of development and technical operations. By making cluster management, process deployment and runtime-scheduling manageable within a live-programming environment demanding real-time resource analysis and optimization becomes possible.

3.4. Visualisation and Analysis

Live programming provides excellent support for real-time data visualisation and analysis. As the language R has been demonstrated so powerfully, a dynamic code environment is very useful for data inspection, analysis and reduction. This ability to both produce simulation data and analyze that data in real-time is a powerful combination in the exploration of new science. As the scale of data being produced by "big science" reaches into the petaflops per second[6] there is an increasing interest in this style of "big data" analysis being managed in real-time. In HPC it is increases the case that big data cannot be moved from the site but must be visualized in situ when the job is running, or afterwards with a separate diagnostic program. Live programming environments, such as Extempore, may provide one possible solution to this problem by coupling highly efficient real-time data processing with the ability to steer the data analysis pipeline with unprecedented flexibility.

3.5. Exploration

By supporting human-in-the-loop interaction live programming supports changes to HPC codes in time scales applicable to direct human perception. Changes made to a code's textual representation, can be immediately integrated into a running system, and the effectiveness of those changes can be directly witnessed by the developer. Increasing the role of human perception in development allows programmers to both *see* and *hear* the changes that are introduced into a running simulation in direct response to modifications made to the programs "static" code representation. This in turn allows a broader range of human sense perceptions to be brought to bear on a given problem domain, increasing the potential for learning and discovery.

[6] As an example, the Square Kilomoter Array is expected to produce in excess of 100 petaflops per second [4]

4. Discussion

Our simple PIC case study has shown that live programming can be leveraged to rapidly prototype *new* simulation models based upon extant HPC codes. Figure 2 highlighted the ease with which a C library can be incorporated into a top level Extempore simulation loop. The accompanying video presentation[7] demonstrates how Extempore can then be used to decompose the simulation across a distributed and potentially heterogeneous cluster with significant levels of control over the codes executing on each node.

Legacy code can be efficiently replaced over time as demand for increased performance, interactivity, numerical accuracy or functionality arises. Figure 6 provided an example of how strong language integration with C made legacy-code integration both convenient and efficient. Strong support for legacy codebases is of vital importance to the Scientific community, not only to support code reuse, but also where the communities believe the legacy code to be strong.

Figure 7 provided a pragmatic example of how simple, but often deep changes can be made to an active simulation — without losing the execution state of the system. In this case a change to the numerical accuracy of the simulation was prototyped through the addition of a simple change to the interpolation algorithm. Experimentation with small and relatively simple changes, such as this one, is sometimes resisted due to the overall complexity involved in releasing new versions of a large monolithic codebase. Live programming also removes the need to restart a running simulation, a significant advantage for long running, resource intensive production environments.

Our small case study has attempted to demonstrate how the Extempore programming language can be used as an interface for human-in-the-loop interactivity. We anticipate that providing scientists with the tools to modify the current execution state of their simulations, by changing and modifying a codes textual representation on-the-fly, will present opportunities for "deep" interaction. Significantly these opportunities exist not just to support *engineering*, as alluded too above, but also to support *science*. Live programming is not simply for code debugging, optimization, and refactoring, although these things are extremely valuable, but also for exploration and discovery. In Figure 5 we demonstrated how the addition of a new external electric field could be *tested* through *experimentation* in the running simulation. We believe that the ease with which these types of *experiments* can be designed, executed, and evaluated, will help to assist scientists to obtain a greater understanding of complex computational models.

Our purpose in this paper has been to draw a picture of what "live programming" might mean for HPC in the future. We have occasionally made remarks in the paper to a near performance equivalence between live programming environments (Extempore in this case) and existing HPC technologies. Although the presentation of detailed performance benchmarks for Extempore against other languages is work in progress, and outside of the scope of this paper we remark that single node benchmarks, we have performed from the "Computer Language Benchmarks Game" [16] for code written entirely in Extempore (i.e. not harnessed from C code), show performance penalties against code written entirely in C of less than a factor of two (1.25 for the Fasta Benchmark with N=25,000,000 and 1.50 for the NBody Benchmark with N=50,000,000). Experiments, that we have performed and that harness an MPI version of our PIC framework, demonstrate comparable performance, as shown in Table 1.

[7] `https://vimeo.com/126577281`

Conclusion

This paper has provided a proof-of-concept demonstration that a live-programming environment can be used with extant HPC codes in a *deeply interactive* fashion. Our approach promises to combine the development experience of writing high-level glue code in a scripting language with the performance and control of native code. By giving scientists and HPC application developers faster feedback about what works, what is broken, and what our codes are doing *while they are running*, we can develop simulation codes more efficiently and make better use of increasingly complex array of HPC resources at our disposal.

In this paper we have not spent time describing the details of the Extempore language and environment. Although Extempore is somewhat unique in its efforts to specifically support "live programming" there is a number of other new general-purpose languages that have recently also begun to target high performance, scientific computing. Swift [6], Rust [14] and Julia [1] are a few of the new languages targeting high performance, parallel, computing contexts with some degree of "liveness" (Swift playgrounds for example). These new languages can be useful for scientific simulation in the future.

In this paper we have highlighted how Extempore may be used to rapidly prototype "deep" interaction by extending extant HPC codes. Ultimately our goal with Extempore is not simply to provide a high performance, hot-swappable C/C++/Fortran but, instead, to work towards the more far-reaching goal of bringing modern language design concepts, such as advanced type systems, into the realm of high performance scientific computing. Our approach has been a pragmatic one that emphasises the value in legacy codes, approaching the wider problem by harnessing such codes and understanding them from the "inside out" as time and needs demand. Clearly the PIC case study described here has lent itself well to harnessing and live-programming. An important consideration for scientific programming in general might be what program architectures and design patterns can be used to enable easy and efficient harnessing by a live-programming environment in the way that has been demonstrated here.

Our codes are open-source and available online, both the original skeleton codes[8] and the "live" version in Extempore.[9] While we have tried in this paper to give a feel for the "practice" of live programming, the static nature of paper publication is a natural limitation. We strongly encourage readers to watch the complementary video[10], which we hope conveys more of the "experience" of live programming.

Acknowledgments

We would like to acknowledge the support of Professor John Taylor, Director of eResearch and Computational Sciences at the Australian Commonwealth Scientific and Industrial Research Organisation (CSIRO), and the Particle in Cell and Kinetic Simulation Center (PICKSC), funded by the US National Science Foundation, NSF Grant ACI-1339893, for their support of this work.

[8] `http://picksc.idre.ucla.edu/software/skeleton-code`
[9] `https://github.com/digego/extempore/tree/master/libs/external/pic`
[10] `https://vimeo.com/126577281`

References

1. Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

2. C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. CRC Press, October 2004.

3. Viktor K. Decyk. Skeleton Particle-in-Cell Codes on Emerging Computer Architectures. *Computing in Science & Engineering*, 17(2):47–52, March 2015.

4. P.E. Dewdney, P.J. Hall, R.T. Schilizzi, and T.J.L.W. Lazio. The square kilometre array. *Proceedings of the IEEE*, 97(8):1482–1496, Aug 2009.

5. Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, December 2007.

6. James Goodwill and Wesley Matlock. The swift programming language. In *Beginning Swift Games Development for iOS*, pages 219–244. Springer, 2015.

7. John Hules and Jon Bashor. Report of the 3rd DOE Workshop on HPC Best Practices: Software Lifecycles. US Department of Energy, 2009.

8. C Lattner and V Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, January 2004.

9. Paulett C Liewer and Viktor K Decyk. A general concurrent algorithm for plasma particle-in-cell simulation codes. *Journal of Computational Physics*, 85(2):302–322, December 1989.

10. Marta Mattoso, Jonas Dias, Kary A. C. S. Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vítor Silva, and Daniel de Oliveira. Dynamic steering of HPC scientific workflows: A survey. *Future Generation Computer Systems*, 2014.

11. Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the r language. *ECOOP 2012–Object-Oriented Programming*, pages 104–131, 2012.

12. Jurriaan D Mulder, Jarke J van Wijk, and Robert van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, February 1999.

13. Travis E. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

14. The Rust Programming Language. `http://www.rust-lang.org`.

15. Andrew Sorensen. Extempore. `http://extempore.moso.com.au/`.

16. The Computer Languages Benchmark Game. `http://benchmarksgame.alioth.debian.org/`.