Supercomputing Frontiers and Innovations

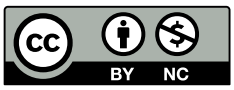
2015, Vol. 2, No. 4

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents



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))))
```

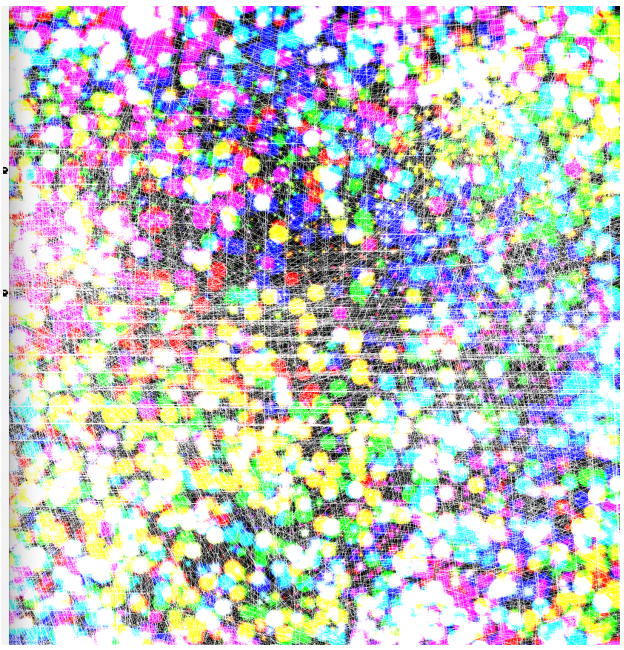


Figure 4. A screenshot of the running PIC simulation with a live visualisation of the particles. Here the visuals are shown next to the Extempore code in the text editor, but they could be shown on any monitors (including being streamed over the network)

(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/`.

DOI: 10.14529/jsfi150402

Creating interconnect topologies by algorithmic edge removal: MOD and SMOD graphs

Marek T. Michalewicz [1, 2, 3], *Łukasz P. Orłowski* [1, 3, 4], *Yuefan Deng* [1, 3, 4]

We introduce a method of constructing classes of graphs by algorithmic removal of entire groups of edges. Our approach to creating new classes of graphs is to focus entirely on the structure and properties of an adjacency matrix. At an initialisation step of the algorithm we start with a complete (fully connected) graph.

In Part I we present MOD and arrested MOD graphs resulting from removal of square blocks of edges at each iteration and substitution of removed blocks with a diagonal matrix with one extra pivotal element along the main diagonal. The MOD graphs possess unique and useful properties. All important graph measures are easily expressed in analytical form and are presented in the paper. Several important properties of MOD graphs are compared very favourably with graphs representing common interconnect topologies: hypercube, 3D and 5D tori, TOFU and dragonfly. This lead us to consider MOD and arrested MOD graphs as interesting candidates for effective supercomputer interconnects.

In Part II, at each iterative step we successively remove triangular shapes from the adjacency matrix. This iterative process leads to the final matrix which has two Sierpiński gaskets aligned along the main diagonal. It will be shown below that this new class of graphs *is not* a Sierpiński graph, since it is the adjacency matrix which has a structure of a Sierpiński gasket, and not a graph described by this matrix. We call this new class of graphs Sierpiński-Michalewicz-Orłowski-Deng (SMOD) graphs. The most remarkable property of the SMOD class of graphs, is that irrespective of the graph order, the diameter is *constant* and equals 2. The size of the graph, or the total number of edges, is about 10% of the size of a complete graph of the same order.

We analyse important graph theoretic characteristics related to the topology such as diameter as a function of graph order, size, mean path length, ratio of the graph size to the size of a complete graph of the same order, and some spectral properties.

Keywords: supercomputer interconnects, big data, exascale computing, graph theory, topology of graphs, classes of graphs, graph generation.

Introduction

The critical design characteristics of the future exascale supercomputer systems will be their interconnect topology, routing algorithms, and connect bandwidths. In this paper we propose a new algorithmic method of constructing well connected graphs, with reasonably small diameter, and low mean path, which may be useful in consideration of future exascale systems. We focus exclusively on the graph theoretic characteristics of the interconnect, hence the analysis of real supercomputer interconnects is restricted to properties derived from graph topology.

During the past 30 years, graphs like hypercube [16], tori [11], and trees [24] have been widely adopted as the topologies of choice for supercomputer networks [18]. More recently, butterfly graphs [22] and dragonfly [13, 19, 21] are entering the market. As the number of computing units in a supercomputer grows to a few million processing cores such as the fastest Tianhe-2 [1, 20] computer in June 2015 with 3.12 million cores, conventional wisdom in selecting a

[1]A*STAR Computational Resource Centre, Singapore 138632, Singapore
[2]A*STAR Institute of High Performance Computing, Singapore 138632, Singapore
[3]Institute for Advanced Computational Science, Stony Brook University, New York 11794-3600, USA
[4]Department of Applied Mathematics and Statistics, Stony Brook University, New York 11794-3600, USA

topology might not lead to the optimal interconnect design. We must leverage on mathematics to discover the state-of-the-art topologies for more efficient networks.

Design of efficient networks requires optimization in multiple dimensions. First, in mathematical dimension, we must design the best network topologies. Second, in engineering dimension, these topologies must be implementable using existing electronic technologies and fit the current architecture standards. Third, in computer science dimension, the wired topologies must be consistent with an efficient routing scheme to generate reliable and fast networks that are easy to use. Finally, in economics dimension, the system must not be too costly. In this report, we focus on optimization of the first dimension, i.e., mathematical aspect. Hence, we are not concerned here with the study of bi-sectional bandwidth, routing tables and routing or physical layout of cables and fibres.

Our method is to explore graph generation techniques that start with adjacency matrices of arbitrary size and subsequently operate solely on adjacency matrices, which yield graphs, rather than generating graphs via diagram constructions first and treating the adjacency matrix as a derived characteristic [15]. Additionally, in our approach we always start with a complete graph of a given order and we proceed by eliminating the edges, never by starting from a smaller order graph and growing it bigger by embedding or other decorations.

We only focus on the topology design, excluding the underlying fabric, therefore the adjacency matrices that we build, and in what follows always denote by A, are symmetric with values of **0** when there is no edge, and **1** when vertices are connected. This translates into unweighted, undirected and connected graphs.

We study graphs whose topologies would potentially yield optimal interconnects for Supercomputers and Big Data systems, hence we turn our attention to the following characteristics:

1. Minimum diameter
2. Good paths distribution
3. Low mean path length
4. Small size, *i.e.* number of edges
5. Small degrees of vertices

Part I

1. Michalewicz-Orłowski-Deng (MOD) Algorithm

We consider graphs $G = (N, L)$, where N is a set of vertices and L is a set of undirected edges (i.e. set of unordered pairs of vertices). The graph order $|N|$, *i.e.* numbers of vertices, is restricted to powers of 2: $|N| = n = 2^m$, $m \in \mathbb{N}$, $m \geq 2$.

Michalewicz-Orłowski-Deng (MOD) algorithm begins with a complete graph and in every step reduces the number of remaining edges by about a half, resulting in a graph with $O(nlog_2n)$ edges. The algorithm converges in $log_2n - 1 = m - 1$ steps.

1.1. The algorithm sequence

The MOD algorithm proceeds through the following sequence of steps:

1. Start with an adjacency matrix of a complete graph
2. Treat the entire adjacency matrix as a single block on the main diagonal

3. Split the block(s) on the main diagonal into quadrant(s) of 4 sub blocks: two diagonal blocks and; upper and lower counter-diagonal blocks
4. Substitute every upper counter-diagonal block with a sum of identity matrix plus pivotal edge
5. Substitute every lower counter-diagonal block with transpose of upper counter-diagonal block of step 4.
6. Repeat steps 3-5 or stop if the size of blocks is 2.
   Mathematically, the algorithm looks in the following way:

$$A = J - I$$

a) Initialization of the MOD algorithm - step 1 above

$$A \leftarrow \left(\begin{array}{c|c} B_d^{(1)} & B_u^{(1)} \\ \hline B_l^{(1)} & B_d^{(1)} \end{array}\right)$$
$$B_u^{(1)} = I + P_e$$
$$B_l^{(1)} = B_u^{(1)T}$$

b) Iteration 1 of the MOD algorithm - steps 2-5 above

$$\vdots$$

$$B_d^{(1)} \leftarrow \left(\begin{array}{c|c} B_d^{(2)} & B_u^{(2)} \\ \hline B_l^{(2)} & B_d^{(2)} \end{array}\right)$$
$$B_u^{(2)} = I + P_e$$
$$B_l^{(2)} = B_u^{(2)T}$$

c) Iteration 2 of the MOD algorithm - step 6 above

$$B_d^{(m-1)} \leftarrow \left(\begin{array}{c|c} B_d^{(m)} & B_u^{(m)} \\ \hline B_l^{(m)} & B_d^{(m)} \end{array}\right)$$
$$B_u^{(m)} = I + P_e$$
$$B_l^{(m)} = B_u^{(m)T}$$

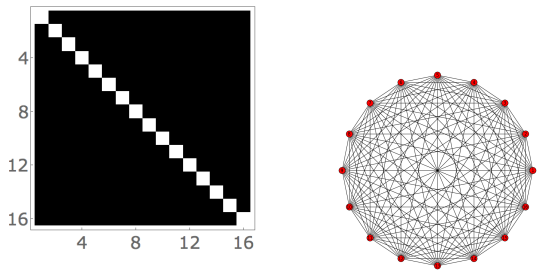
d) Final, m^{th} iteration of the algorithm.

Figure 1. Sequence of steps of the MOD algorithm

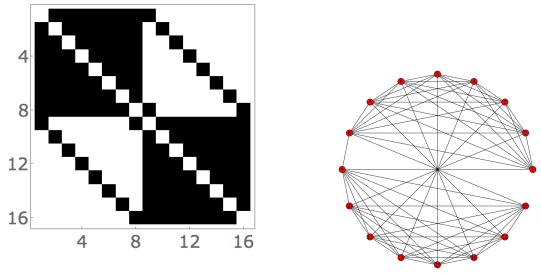
here J is an all-ones matrix, I is an identity matrix, $B_d^{(p)} \in \mathbb{R}^{k\times k}$ are a diagonal block in the p^{th} step of the algorithm, $P_e \in \mathbb{R}^{k\times k}$ is the pivotal element matrix defined as $(P_e)_{ij} = \begin{cases} 1 & i = k, j = 1 \\ 0 & \text{otherwise} \end{cases}$
$B_u^{(p)}$and $B_l^{(p)}$ are respectively upper and lower counter-diagonal blocks in the p^{th} step of the algorithm and $k = \frac{n}{2^p}$.

1.2. Visualisation of steps of the MOD algorithm

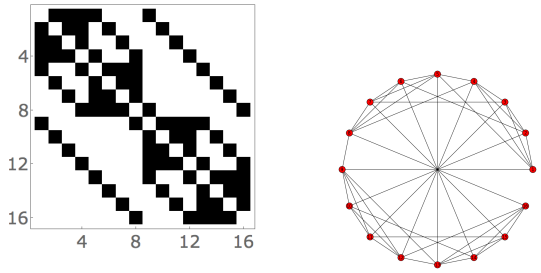
The figure below illustrates adjacency matrices generated in every step of the MOD algorithm (on the left) and the diagrams of graphs with a group of edges removed at every step (on the right).



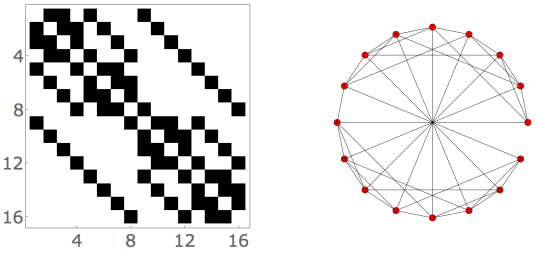
a) Initialization



b) Iteration 1



c) Iteration 2

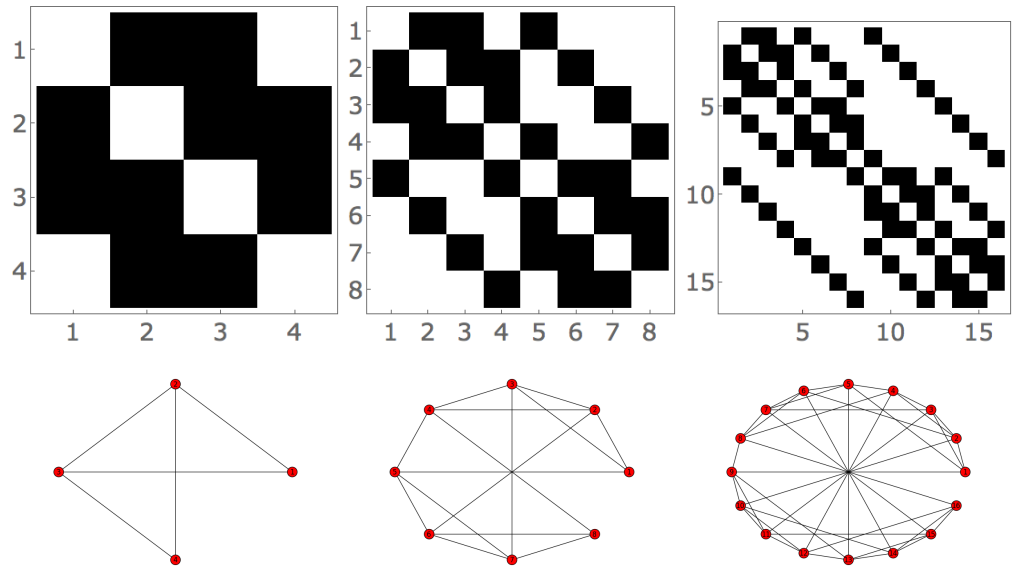


d) Iteration 3, final

Figure 2. Steps of the MOD algorithm generating MOD graph with 16 vertices

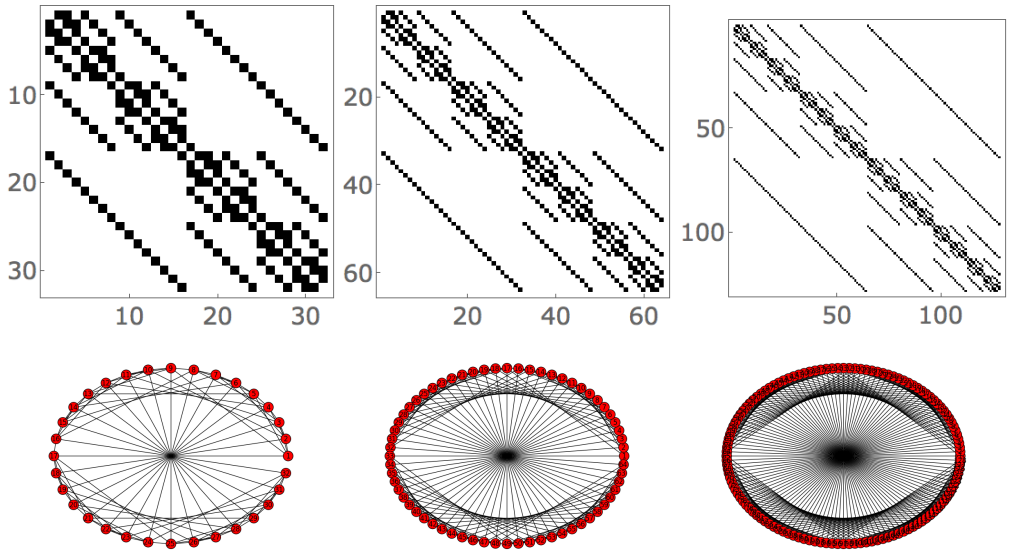
1.3. Visualisation of low order MOD graphs

The following figure exemplifies adjacency matrices generated by the MOD algorithm, along with corresponding graphs representation.



a) 4 vertex MOD **b)** 8 vertex MOD **c)** 16 vertex MOD

Figure 3. MOD graphs and their adjacency matrices for small $|N| = n = 2^m$ graphs with $2 \leq m \leq 4$



a) 32 vertex MOD **b)** 64 vertex MOD **c)** 128 vertex MOD

Figure 4. MOD graphs and their adjacency matrices for small $|N| = n = 2^m$ graphs with $5 \leq m \leq 7$

2. Properties of MOD graphs

In this section we collect results for several basic MOD graph properties such as the diameter, mean path length, size of the graph (*i.e.* number of edges), the ratio of MOD graph size to complete graph size of the same order, and the vertex degrees. It turns out that most of these properties can be derived analytically and are expressed by simple relationships. We also study the spectra of adjacency matrices and attempt to relate them to other graph properties.

2.1. Diameter and mean path length

Two very important properties of graphs are the diameter and mean path length. The diameter of a graph is the greatest distance between any two vertices in a graph, where distance between two vertices is defined as the shortest of all paths connecting these vertices.

The diameter of MOD graphs is found to be

$$d_{MOD}(n) = log_2 n = m, \qquad \text{for } 2 \leq m \leq 3 \tag{1}$$

$$d_{MOD}(n) = log_2 n - 1 = m - 1, \quad \text{for } m \geq 4 \tag{2}$$

The mean path length, l_G, is the average of distances between any two vertices in a graph, defined as follows

$$l_G = \frac{1}{n(n-1)} \sum_{i \neq j} d(v_i, v_j) = \frac{1}{n(n-1)} \sum_{i \neq j} (D)_{ij} \tag{3}$$

where $d(v_i, v_j)$ denotes distance between i^{th} and j^{th} vertex and D denotes a distance matrix corresponding to a graph G. The latter formula provides a prescription for computation of l_G.

The mean path length of the MOD graphs converges to half of the diameter, for large n.

$$\lim_{n \to \infty} \left(\frac{1}{n(n-1)} \sum_{i \neq j} (D)_{ij} \right) = \frac{1}{2} d_{MOD}(n) = \frac{1}{2}(m-1) \tag{4}$$

The figure below depicts a diameter and mean path of the MOD graphs as a function of graph order n.

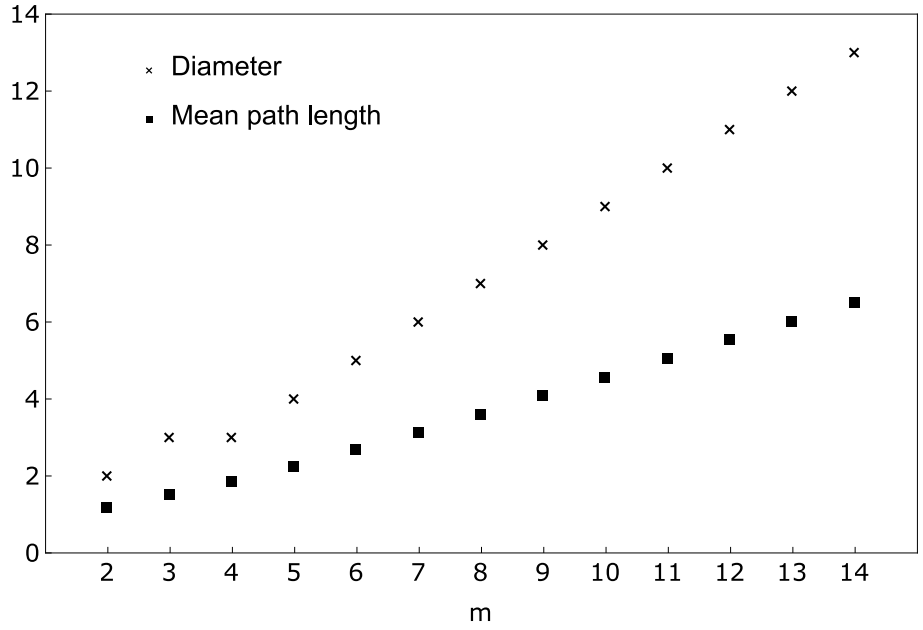


Figure 5. Diameter and mean path of the MOD graphs as a function of graph order n

2.2. Size of MOD graphs and their vertex degrees

Theorem 1. The size of a MOD graph of order n is given by:

$$\begin{aligned} |L| &= \frac{n}{2}(log_2 n + 1) - 1 \\ &= (m+1)2^{m-1} - 1 \end{aligned} \tag{5}$$

Where $n = 2^m$

Proof. We prove the formula by counting the edges removed at every step of the algorithm.

$$\begin{aligned} |L| &= \frac{1}{2}\left[n(n-1) - 2(\frac{n}{2})^2 + n + 2 - 2^2(\frac{n}{2^2})^2 + n + 2^2 + ... - 2^m(\frac{n}{2^m}) + n\right] \\ &= \frac{1}{2}\left[n(n-1) - n^2\sum_{i=1}^{m}\frac{1}{2^i} + nm + \sum_{i=1}^{m-1}2^i\right] = \frac{1}{2}\left[nm + 2n^2 - n - n^2\sum_{i=0}^{m}\frac{1}{2^i} + \sum_{i=0}^{m-1}2^i - 1\right] \\ &= \frac{1}{2}\left[nm + 2n^2 - n - \frac{n^2}{2^m}(2^{1+m} - 1) + 2^m - 2\right] = \frac{1}{2}\left[nm + 2n^2 - n - 2n^2 + \frac{n^2}{2^m} + 2^{1+m} - 2\right] \\ &= \frac{n}{2}(log_2 n + 1) - 1. \end{aligned}$$

□

The following figure shows how the size of the MOD graphs increases as a function of the graph order $n = 2^m$ grows from 4 to 16,384 (or m grows from 2 to 14).

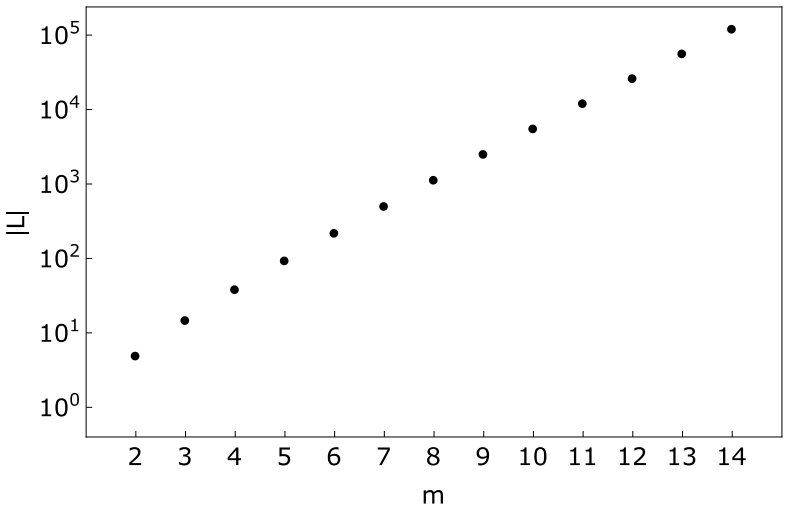


Figure 6. Size of MOD graphs for $2 \leq m \leq 14$

All the vertices of the MOD graph have a constant degree $m + 1 = log_2 n + 1$, except for two vertices, numbered 1^{st} and n^{th} in adjacency matrices which are of degree $m = log_2 n$. This property can be expressed in the following way:

$$Au = (m, m + 1, ..., m + 1, m)^T. \tag{6}$$

where A is the adjacency matrix and $u^T = (1, \ldots, 1)$

2.3. Bisections of MOD graphs

Bisection is a subset of edges, that, when removed, separates a graph into two disconnected components of equal order (or almost equal for graphs of odd order). In the network theory, the quantity of interest is the bisection bandwidth, which can be derived from minimum bisection (a graph theoretic property) by multiplying each edge from the cut by its weight (or bandwidth in the network theory language). Here we assume that each edge has weight equal 1, which provides the best basis for topology comparisons. In this section we look at the minimum bisection of MOD graphs. We do not formally derive the formula for the number of edges in the minimum bisection, but instead arrive at the formula after a discussion to give the intuition of what is the minimum bisection of a MOD graph.

Let us observe, that every vertex v_k in a MOD graph of order n is directly connected to a vertex $v_{(k+n/2) \mod n}$. Furthermore every vertex v_k is connected to v_{k-1} and v_{k+1}, except for v_1 and v_n. This means that the minimum bisection (*i.e.* a bisection with the minimum number of edges) has to contain all the edges connecting the v_k with $v_{(k+n/2) \mod n}$ for $1 \leq k \leq n/2$ plus one edge connecting $v_{n/2}$ and $v_{n/2+1}$. This is presented in the following figure:

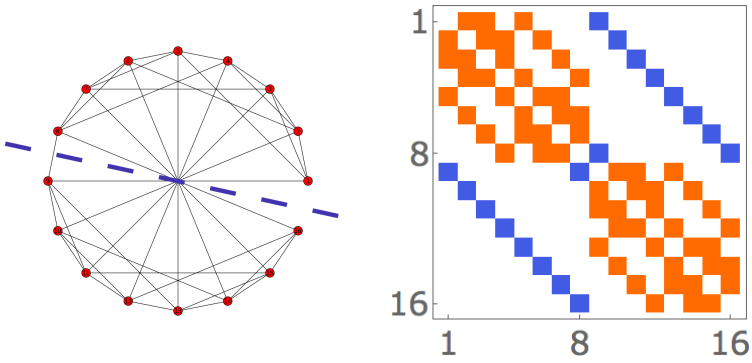


Figure 7. Minimum bisection of MOD graph of order 16 presented in graph (the cut in blue) and its adjacency matrix (entries in blue)

This leads us to an observation that the number of edges in the minimum bisection in a MOD graph is

$$|\min_{bisections} B(V_1, V_2)| = \frac{n}{2} + 1 = 2^{m-1} + 1 \tag{7}$$

Where V_1 and V_2 are two disjoint sets of vertices of equal size, whose sum is V.

In terms of graphs suited for topologies of interconnects, the higher the size of the minimum bisection, the better, meaning that the network based on such topology allows more data to flow from one half of the network to the other under uniform traffic pattern [18].

The following table summarises the minimum bisections and some other parameters of topologies that most resemble MOD graphs.

Table 1. Formulas for most important parameters including bisection width of hypercube, tori and MOD. Note that MOD has two vertices of degree m as depicted in (6)

Topology	Order	Degree	Diameter	Bisection
Hypercube	2^m	m	m	2^{m-1}
3D Torus	m^3	6	$3m/2$	$2m^2$
5D Torus	m^5	10	$5m/2$	$2m^4$
MOD	2^m	$m+1$	$m-1$	$2^{m-1}+1$

2.4. Edge ratio of MOD graphs

Definition 1. Edge ratio of a graph $G = (N, L)$ is a ratio of the size of a graph, $|L|$, to the size of a complete graph of the same order, i.e.:

$$L_{ratio} = \frac{2|L|}{|N|(|N|-1)}. \tag{8}$$

The edge ratio of the MOD graphs follows immediately from (5) and is given by:

$$L_{ratio} = \frac{nlog_2 n + 2n - 2}{n^2 - n} \tag{9}$$

$$= \frac{(m+2)2^m - 2}{2^m(2^m - 1)} \tag{10}$$

In the limit of large n an edge ratio of the MOD graph converges to 0, *i.e.*

$$\lim_{n\to\infty} L_{ratio} = 0 \tag{11}$$

2.5. Eigenvalues of the MOD graphs

Although the interpretation and meaning of *all* eigenvalues of adjacency matrices is yet unclear and subject to the study of spectral graph theory [17, 25], there are a few properties of MOD graphs which we understand and discuss here. The distribution of eigenvalues for MOD graphs with $2 \leq m \leq 14$ is represented below:

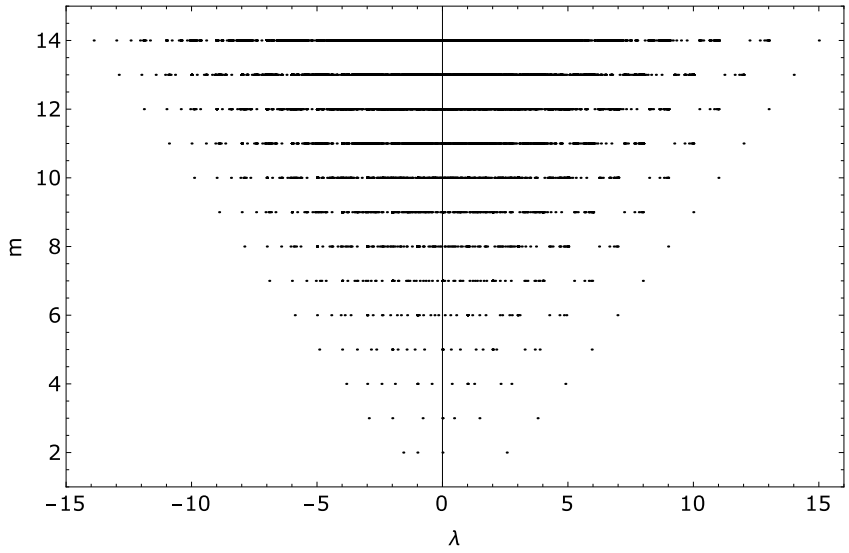


Figure 8. Distribution of eigenvalues for MOD graphs with $2 \leq m \leq 14$.

We denote the greatest eigenvalues as λ_1 and the least one as λ_n. We also observe that the average vertex degree is bounded by λ_1

$$\frac{1}{n}u^T A u = \frac{1}{n}\sum_{i,j}^{n}(A)_{ij} = 2\frac{|L|}{|N|} < \lambda_1 \tag{12}$$

and approaches λ_1 as the order of graph grows large. It follows trivially from Eq. 5 that

$$\lim_{n\to\infty} \lambda_1 = \lim_{n\to\infty} 2\frac{|L|}{|N|} = m + 1 \tag{13}$$

Furthermore, one sees that

$$\lambda_{n-1} = -m + 1 \tag{14}$$

$$\lambda_n \approx -m \tag{15}$$

We observe from Eq. 12 and Eq. 14 that for MOD graphs the greatest eigenvalue is equal to the spectral radius i.e. $\lambda_1 = \rho(A(G))$. The eigenvalues are bound as follows

$$-m < \lambda_i < m + 1, \text{ for all } i \in [1, n] \tag{16}$$

Finally, since: i) trace of the matrix is invariant with respect to similarity transformations, and, ii) the sum of all eigenvalues of an adjacency matrix of a graph with no self-loops is 0: $tr(A) = 0$.

$$\sum_{i=1}^{n} \lambda_i = 0 \tag{17}$$

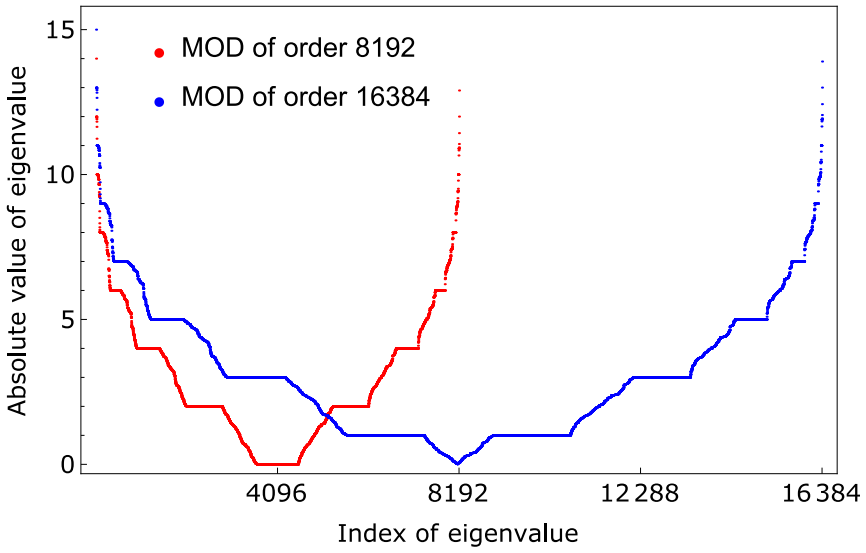
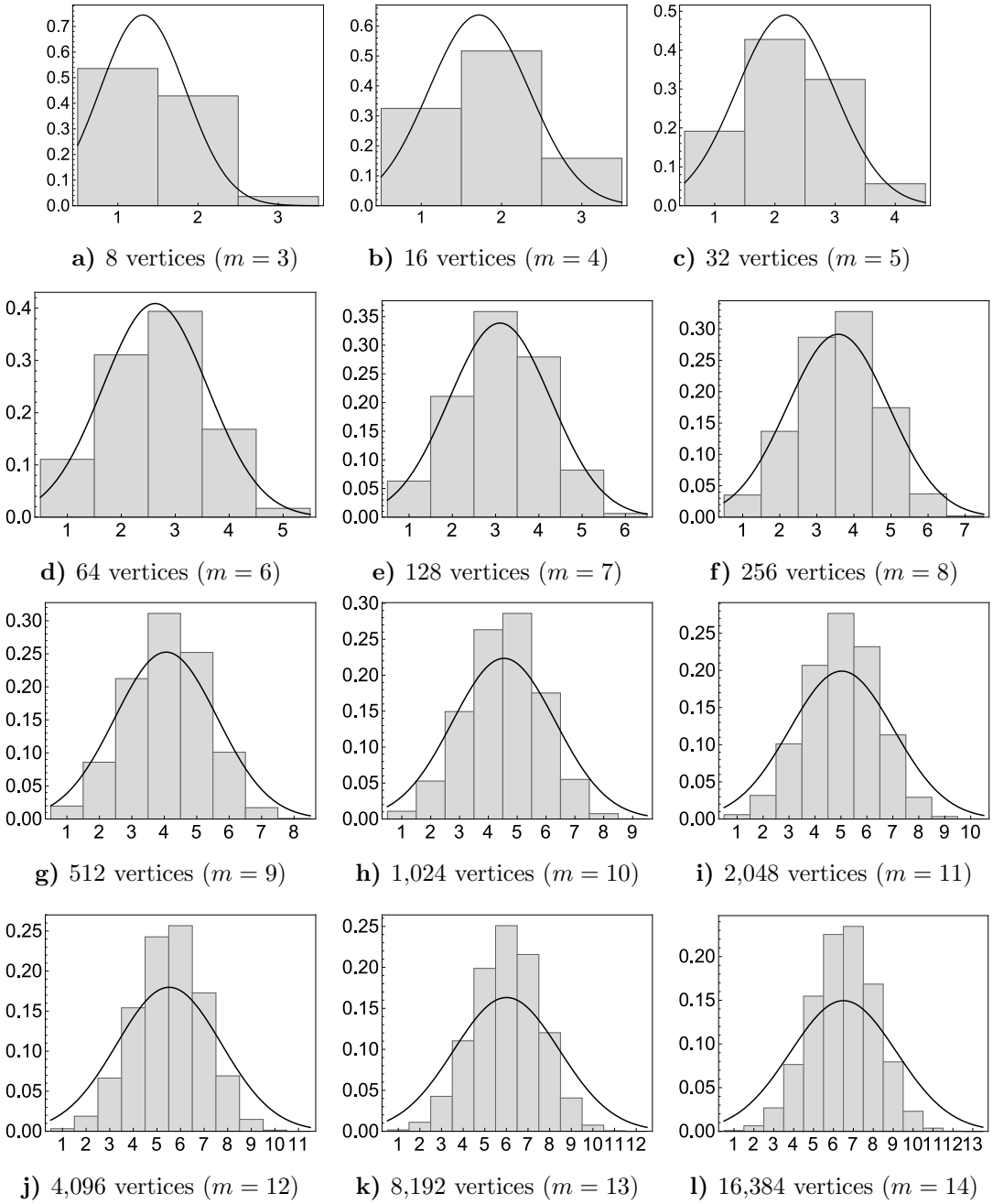


Figure 9. Absolute values of eigenvalues for the MOD graphs of order $n = 2^{13}$ (red) and $n = 2^{14}$ (blue). NB For $j \geq \frac{n}{2}$, $\lambda_j \leq 0$

The Figure 7 above represents the absolute values of all eigenvalues of the MOD graphs of order $n = 2^{13}$ (red) and $n = 2^{14}$ (blue). It is clearly seen that there are regions of the spectra where all eigenvalues assume natural number value, and are highly degenerate. For m odd, the integer eigenvalues are even, for m even, they are odd. Another fact that can be observed from the plots, is that very nearly half of eigenvalues are negative and half positive. The minimum of the absolute value of eigenvalues is non-degenerate for even values of m and degenerate for odd values of m and in both cases falls at or is centred (respectively) at $n/2$. For odd values of m the minimum of the absolute value of eigenvalues is equal zero, and for even m is slightly above zero.

2.6. Distance distribution of MOD graphs

The histogram of all distances expresses the discrete probability distribution of a distance between any two vertices in a graph. This distribution is an essential signature of a graph. It relates to the mean path length and the maximum distance, *i.e.* the graph diameter. In the context of practical supercomputer interconnect topologies distance distribution translates to network latency between the given pair of ports in the network. There are important implications of distribution of distances on the graph traversal, which in terms of network topologies translate to flow control and routing algorithms.



a) 8 vertices ($m = 3$) **b)** 16 vertices ($m = 4$) **c)** 32 vertices ($m = 5$)

d) 64 vertices ($m = 6$) **e)** 128 vertices ($m = 7$) **f)** 256 vertices ($m = 8$)

g) 512 vertices ($m = 9$) **h)** 1,024 vertices ($m = 10$) **i)** 2,048 vertices ($m = 11$)

j) 4,096 vertices ($m = 12$) **k)** 8,192 vertices ($m = 13$) **l)** 16,384 vertices ($m = 14$)

Figure 10. Distribution of distances in MOD graphs with $4 \leq m \leq 14$

Figure 10 depicts histograms of distances in MOD graphs of order $n = 2^m$, for $4 \leq m \leq 14$ overlaid with probability density functions of normally distributed random variable with the same mean value and variance as respective samples. Although at first it might seem that distances are normally distributed - they are not. The histograms of distances in MOD graphs have heavier tails than normal distribution.

A normal distribution (and centralised distributions) results in very predictable distances, meaning that the expected value of path length is equal (or close) to mean path length. Right skewed distributions (those with positive skewness or third standardised moment) have majority of short paths, a desirable characteristic, which may give a performance boost to a routing algorithm, however the vertices which are far away from each other will have negative impact on global communication *e.g. gather and scatter*. Left skewed distributions (with negative skewness) might intuitively seem undesirable. However, for topologies with a very short diameter, such as the dragonfly topology, skewness is not a decisive feature which might disqualify the topology from being a very effective solution.

The most important final point here is that irrespective of the underlying topology, and how good the distance distribution might seem, the ultimate criterion is a communication pattern *within* the algorithm, and how well it matches the hardwired supercomputer network topology. This topic deserves entirely separate treatment. Of course, some common stencils and known implemented algorithms might map very well on standard supercomputing interconnect topologies (see Section 5).

3. Graphs resulting from halted MOD algorithm: arrested MOD graphs

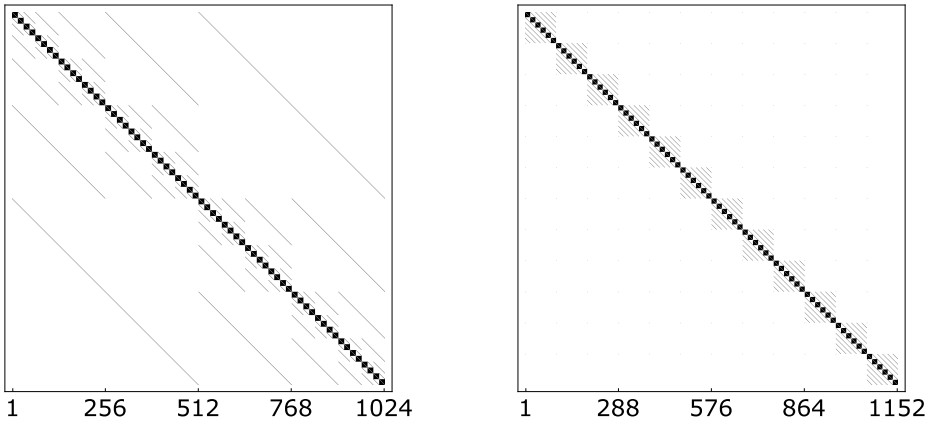
One very attractive feature of the graph creation algorithms presented here, and especially the MOD algorithm, is an ability to halt the iterative process at any step $0 \leq c \leq m-1$ to create what we define as **arrested MOD graphs**, or **aMOD**, in short.

We introduce notation to describe arrested MOD graph of order $n = 2^m$ and halted at c^{th} iteration as $aMOD(m, c)$, for example, if $m = 10$ and $c = 6$, we write: $aMOD(10, 6)$ and it describes the arrested MOD graph of order $n = 2^{10} = 1024$ and halted at the 6^{th} iteration resulting in all-to-all fully connected blocks of $2^{m-c} = 16$ vertices.

This process leads to creating 2^c complete sub-graphs of order 2^{m-c}, or in the language of graph theory 2^c of 2^{m-c}*-vertex cliques*.

Why is this important? It turns out that the most efficient supercomputer interconnect topology (currently) on the market is the dragonfly topology [13, 19, 21], exemplified by the Cray XC30 Aries topology[5]. It is complete at back-plane level, or at what we would attribute in the aMOD context (in our aMOD topology) to $2^{m-c} = 16$. Please note that for the sake of simplification we consider Aries NIC as the end-node (vertex) in a dragonfly topology. This is partially justifiable since all connections between Aries and CPUs run as printed circuit lines, and are not realised as copper or optic links. The adjacency matrices of the $aMOD(10, 6)$ graph and the dragonfly graph of order $n = 1152$ are presented below.

[5] see Table 2



a) Adjacency matrix of $aMOD(10, 6)$ graph, and

b) .. the dragonfly graph of order $n = 1152$.

Figure 11. Comparison of adjacency matrices of aMOD and dragonfly

N.B. The plot of dragonfly contains the whole set of separate connections (dots on our representation of the adjacency matrix) which are too tiny to be resolved in this picture. In contrast, aMOD has connections which run in off-diagonal arrangement and are more clearly visible. Further discussion of similarities and differences is presented in Section 5.

Figure 10 below illustrates how the MOD algorithm for the MOD graph of order $n = 1024$ or $m = 10$, which is halted at every iteration $c = 0, ..., 9$

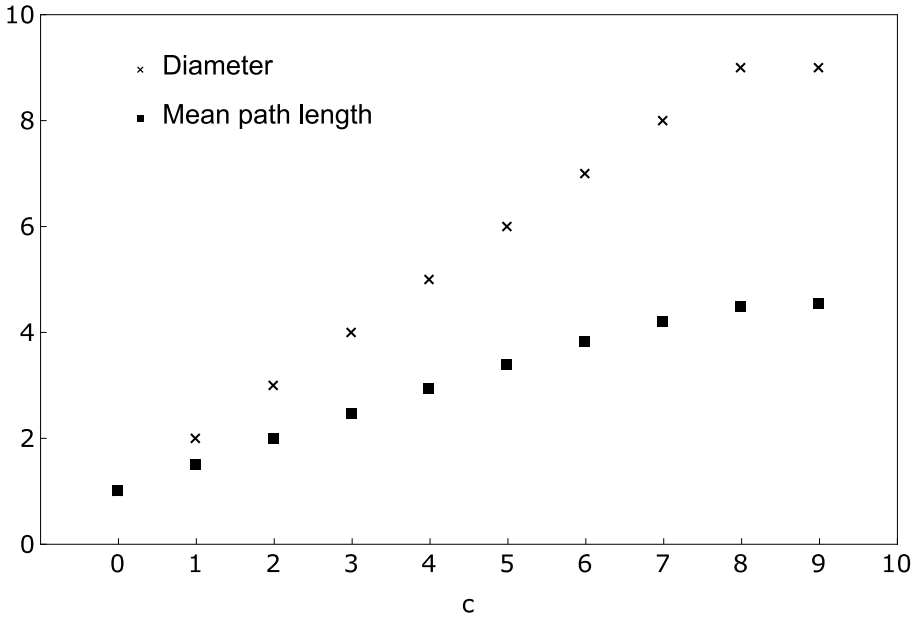


Figure 12. Diameter and mean path length of the aMOD(m,c) graph vs. c; m=10, c-iteration step

The arrested algorithm results in one complete graph ($c = 0$), seven aMOD graphs for $c = 1, 2, ..., 8$, and a saturated MOD algorithm, MOD graph for $c = 9$.
Please note that the adjacency matrices and aMOD graphs for $m = 4$ and $c = 0, 1, 2, 3$ were represented in Figure 2.

The mean path length is about half of the graph diameter, and what is clearly seen, the interesting property of the arrested aMOD graphs - the diameter of the aMOD(m,c) graph is:

$$d_{aMOD(m,c)} = c + 1 \quad 1 \leq c \leq m - 2 \tag{18}$$

$$d_{aMOD(m,c)} = c \quad c = m - 1 \tag{19}$$

The topologies of aMOD graphs could be utilised in the future highly connected and very dense systems, where lot more nodes could be tightly interconnected in all-to-all manner. There is, of course, a price to pay for all-to-all communication: the number of links increases and this adds substantial costs to the entire system cost. However, we believe that the future, ultra-dense systems with on-chip interconnects will allow this type of topology to be implemented. Figure 11 below illustrates the exponential decrease of the size of the aMOD graph as a function of iterations (translated to fully connected blocks (*cliques*) of sub-systems).

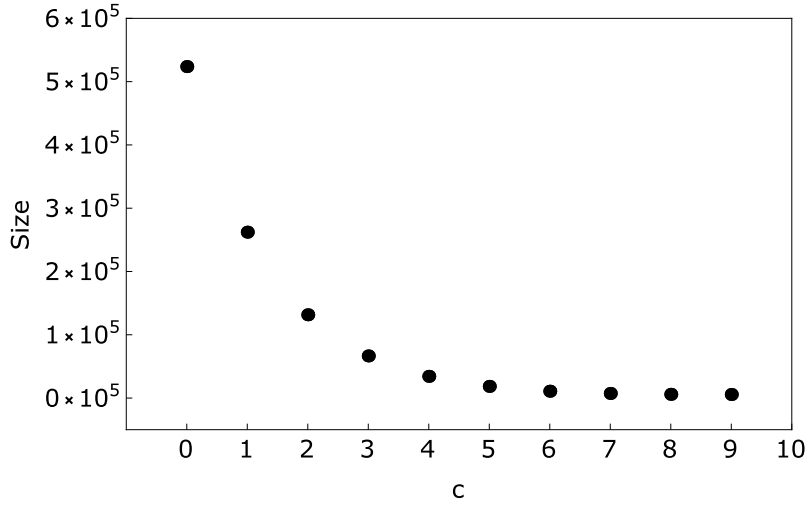


Figure 13. Size of the aMOD(m,c) graph vs. c; m=10, c is the iteration step

4. How all this fits to real Supercomputer interconnect topologies?

The most powerful supercomputer in the world in June 2015 [2], Tianhe-2 has more than 3,000,000 cores. It is well recognised that with limits imposed on increases of clock speed of the new, more powerful systems, the most obvious way to greater computational performance is by increase of parallelism, or in other words, the number of separate computational units: cores in processors or accelerators and a number of nodes in a system. This prospect immediately leads to incredible increase of complexity of possible interconnect networks. Again, the importance of communication, data placement (to minimise communication) and the requirement for robust and well performing network topologies has been well documented [9, 10, 23].

Table 2. Comparison of basic graph properties of MOD and aMOD graphs with graph topologies used in the world's leading supercomputer systems and for recently proposed SlimFly [14] for three different graph orders.
*Note that MOD and aMOD graphs have 2 vertices, whose degree is lower by 1 than that of the remaining vertices - see (6).
**Dragonfly topology as implemented in Cascade has 20 edges per Aries within an electrical group (a two-cabinet arrangement) plus at least one link connecting to a different electrical group

Topology	Graph Order/ Dimensions	Order	Size	Deg.	Diam.	Mean path length	Bisec.
MOD	**1,024**	**1,024**	**5,631**	**11***	**9**	**4.55**	**513**
aMOD	**1,024;16**	**1,024**	**10,815**	**22***	**7**	**3.82**	**513**
Dragonfly	1,152	1,152	11,586	≥21**	5	4.31	12
SlimFly	q = 23	1,058	18,515	35	2	1.97	6,095
3D torus	$8 \times 8 \times 16$	1,024	3,072	6	16	8.01	128
5D torus	$4 \times 4 \times 4 \times 4 \times 4$	1,024	5,120	10	10	5.00	512
Hypercube	10D	1,024	5,120	10	10	5.00	512
TOFU	$3 \times 4 \times 8 \times 2 \times 3 \times 2$	1,152	5,760	10	10	5.38	288
MOD	**2,048**	**2,048**	**12,287**	**12***	**10**	**5.03**	**1,025**
aMOD	**2,048;16**	**2,048**	**22,655**	**23***	**8**	**4.29**	**1,025**
aMOD	**2,048;32**	**2,048**	**37,951**	**38***	**7**	**3.86**	**1,025**
Dragonfly	2,400	2,400	24,300	≥21**	5	4.43	156
SlimFly	q = 31	1,922	45,167	47	2	1.98	14,927
3D torus	$8 \times 16 \times 16$	2,048	6,144	6	20	10.00	256
5D torus	$4 \times 4 \times 4 \times 4 \times 8$	2,048	10,240	10	12	6.00	512
Hypercube	11D	2,048	11,264	10	11	5.50	1,024
TOFU	$5 \times 5 \times 8 \times 2 \times 3 \times 2$	2,400	12,000	10	11	6.07	600
MOD	**4,096**	**4,096**	**26,623**	**13***	**11**	**5.52**	**2,049**
aMOD	**4,096;16**	**4,096**	**47,359**	**24***	**9**	**4.78**	**2,049**
aMOD	**4,096;32**	**4,096**	**77,951**	**39***	**8**	**4.34**	**2,049**
aMOD	**4,096;128**	**4,096**	**270,367**	**133***	**6**	**3.41**	**2,049**
Dragonfly	4,032	4,032	41,181	≥21**	5	4.48	441
SlimFly	q = 37	2,738	75,295	55	2	1.98	25,382
3D torus	$16 \times 16 \times 16$	4,096	12,288	6	24	12.00	512
5D torus	$4 \times 4 \times 4 \times 8 \times 8$	4,096	20,480	10	14	7.00	1,024
Hypercube	12D	4,096	24,576	10	12	6.00	2,048
TOFU	$6 \times 7 \times 8 \times 2 \times 3 \times 2$	4,032	20,160	10	13	6.88	1008

In Table 2 we compare our MOD and aMOD graph topologies with the most common topologies implemented in the largest and most powerful supercomputers in the world: 3D torus

exemplified by Gemini interconnect in Cray XE6 and XK7 [3, 4], 5D torus implemented in IBM BlueGene/Q [5], hypercube partially implemented in SGI Pleiades Ice X at NASA [6, 16]; and Fujitsu K-computer with TOFU proprietary interconnect [12]. Dragonfly network is exemplified by Cascade/Aries technology implemented in Cray XC 30 [19]. Currently the largest Cray XC 30 in the world is Piz Daint at Swiss National Supercomputer Center [7]. Its theoretical peak performance R_{peak} is close to 7.8 PFLOPS and actual peak performance R_{max} is 6.2 PFLOPS. Piz Daint has 5,272 nodes, but each node has one Aries NIC and there are four nodes per Aries. Hence there are 1,318 Aries switches, or in our representation of dragonfly topology, 1,318 vertices in the graph representing Piz Daint supercomputer. It means that a graph representing the 6^{th} most powerful supercomputer in the world according to June 2015 Top500 list [2] has the mean path length ~ 4.3.

We have included arrested aMOD(10,6) graph which is the most similar to dragonfly in terms of clique size and graph size. It is interesting to note that our aMOD(10,6) and aMOD(11,7) graphs have less links (smaller size) than the corresponding dragonfly graphs and also slightly better mean path length. This should perhaps translate to slightly better computational performance, if aMOD topologies were tested, subject to network bandwidth and efficiency of routing algorithms comparable to Cascade systems. This superiority is regained by dragonfly graph topology for a network of order 4,096 or higher. Note, that the the two largest Cray XC series systems ever built: Piz Daint at CSCS and Shaheen II at KAUST [8] have about 1,536 Aries routers, which are treated in our analysis as the vertices of the Cray's implementation of the dragonfly topology interconnect. Hence the largest currently implemented dragonfly topology with 1,536 vertices are well within the orders of graphs analysed and presented in Table 2.

For comparison we have also included SlimFly [14] topology which has a very small diameter of 2 and attracted a lot of attention lately. From our analysis, it is clear that SlimFly does not scale well since its size is based on prime numbers. There are no HPC systems built with interconnect based on SlimFly topology yet. Of course SlimFly significantly outperforms both MOD and aMOD graphs with respect to diameter and minimum bisection, but this comes at the price of very high degree and size (see Table 3).

The second very important observation is that all other interconnect topologies perform rather poorly if we compare them on just two scores: diameter and mean path length for the same or very similar order. On the other hand, they do have substantially smaller size (less edges) - which normally translates to cheaper system. However - that is not a fair comparison with dragonfly topology of Cray XC 30 - as we noted previously we do not count all the cores of the four nodes per Aries - the more adequate comparison between various supercomputers would need to include exact core count, routing technology and total bisectional bandwidth.

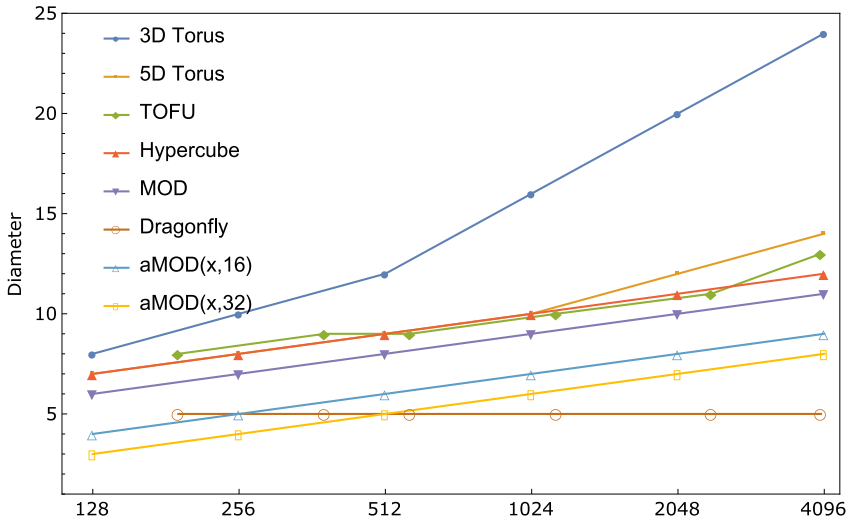


Figure 14. Diameters of MOD graphs and of graph topologies used in the world's leading supercomputer systems as function of graph order

Figure 13 above illustrates the diameter of a graph as a function of its order. This is an important feature in practical interconnect design considerations, since a graph diameter is the upper bound for all distances in a graph. The smaller the diameter, the better, especially for the systems designed for big data, where a program has no control over the placement and locality of data. In 3D torus the diameter grows quite fast and we would expect this topology to be good for a special class of well structured problems with good data locality and very close communication among nearest neighbours, such as regular 3D mesh problems. The diameters of 5D torus and hypercube overlap in the plot since they are the same for graph orders up to 1,024. For larger orders 5D torus diameter is larger than that of hypercube. TOFU performs slightly better than 5D torus and hypercube only for graphs of orders between 512 and 1,024, outside of this range it has larger diameter than both 5D torus and hypercube. Dragonfly has a constant diameter of 5, which is an important property of that graph topology. MOD graphs exhibit moderate growth of diameter as a function of the graph order. MOD graphs' diameters are uniformly better (by 1) than the diameters of hypercube (see Formula 4).

Figure 14 below shows mean path lengths as functions of the order of graphs. Mean path length of 3D torus, 5D torus and hypercube is always half of their diameter, which follows from the periodic boundary conditions of torus (hypercube is a special case of a torus), therefore a rate of growth of mean path length for those three topologies is exactly the same as that of their diameters. Similar behaviour can be observed for TOFU, however its mean path length is slightly more a half of its diameter. MOD graphs have lower diameter than all other graph topologies considered here, bar dragonfly, and the mean path length is expressed by Eq. 3 (in the limit of large graph order). Dragonfly shows a different behaviour. As the order of Dragonfly grows, so does the mean path length, approaching the constant diameter of 5 asymptotically (see Figure 14 and Table 1).

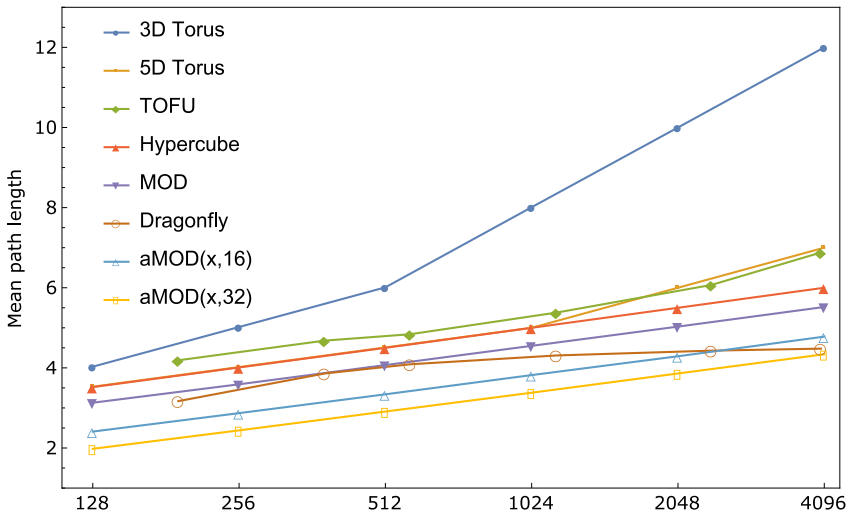


Figure 15. Mean path lengths of MOD graphs and graph topologies used in the world's leading supercomputer systems as function of graph order

The sizes of graphs representing typical supercomputer interconnects are shown in Figure 14. Sizes of graphs grow exponentially, as their orders grow. For dragonfly, the lowest diameter and mean path length is achieved with additional cost of rapid growth of size. MOD and hypercube have similar rate of growth of their size. MOD graphs have more edges though. 5D torus and TOFU both have similar size as the order grows, since they have the same constant vertex degrees. Finally 3D torus whose diameter grows most rapidly, has the slowest rate of size growth.

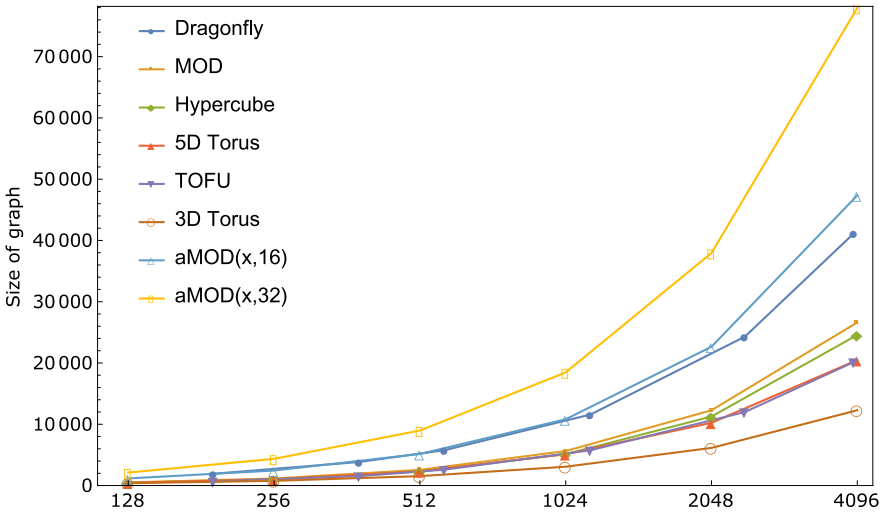


Figure 16. Sizes of MOD graphs and graph topologies used in the world's leading supercomputer systems as function of graph order

Part II

In Part I we presented MOD and arrested MOD graphs resulting from removal of square blocks of edges at each iteration and substituting them with a diagonal matrix with one pivotal element along the main diagonal. Here, we follow similar edge removal strategy, but instead of square shapes we remove triangular shapes from the adjacency matrix, which leads to the final matrix which has two Sierpiński gaskets aligned along the main diagonal. It will be shown below that a graph belonging to this new class of graphs *is not* a Sierpiński graph, since it is the adjacency matrix which has a structure of a Sierpiński gasket, and not a graph described by this matrix. We call this new class of graphs Sierpiński-Michalewicz-Orłowski-Deng (SMOD) graphs.

5. Sierpiński-Michalewicz-Orłowski-Deng (SMOD) Algorithm

Again, as in Section 1, we consider undirected, unweighted graphs. This time, for simplicity of algorithm construction, we restrict the order of SMOD graphs to odd numbers expressed as $|N| = n = 2^m + 1, m \in \mathbb{N}_+$.

The SMOD algorithm is also based on edge removal. The algorithm starts with a complete graph and edges are removed with every iteration, leaving $3^{\log_2(n-1)} = 3^m$ edges upon finishing. The algorithm finishes in $m - 1$ steps.

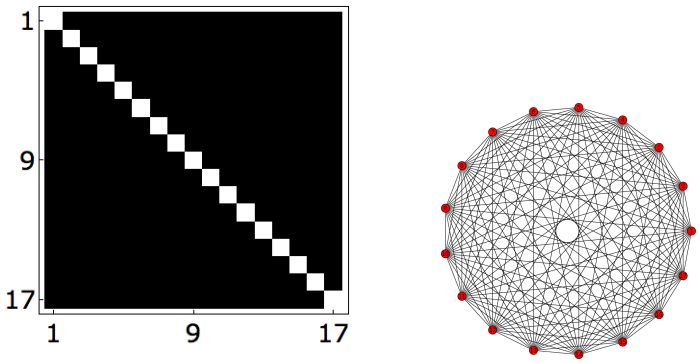
5.1. The SMOD algorithm

The SMOD algorithm has the following steps:

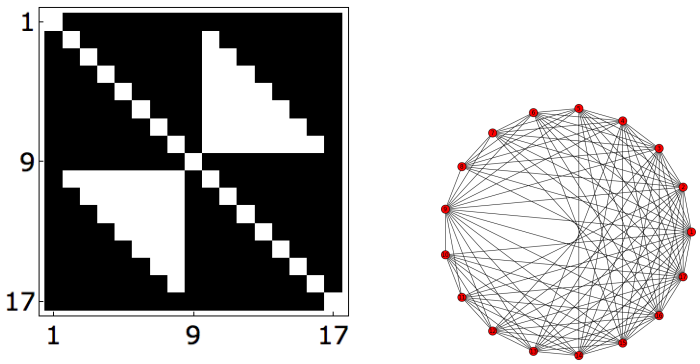
1. Start with an adjacency matrix of a complete graph of order $n + 1 = 2^m + 1$ for $m \geq 2$.
2. The diagonal divides the matrix into two triangular sections: upper-right triangular one (UR) and, lower-left triangular one (LL). We follow exactly Sierpiński iterative process within each of the triangles, with the following qualification: the triangular shapes (matrices) are not equilateral but isosceles right angled triangles.
3. At p^{th} step we subtract $2 \cdot 3^{p-1}$ triangular shapes.
4. Repeat steps 2 and 3 for remaining regions.
5. Stop when triangles reduce to single entry.

5.2. Visualisation of steps of the SMOD algorithm

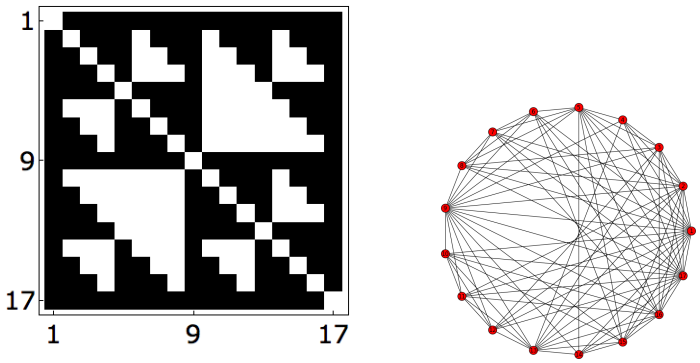
The figure below illustrates adjacency matrices generated in every step of the SMOD algorithm and the diagrams of graphs with a group of edges removed at every step.



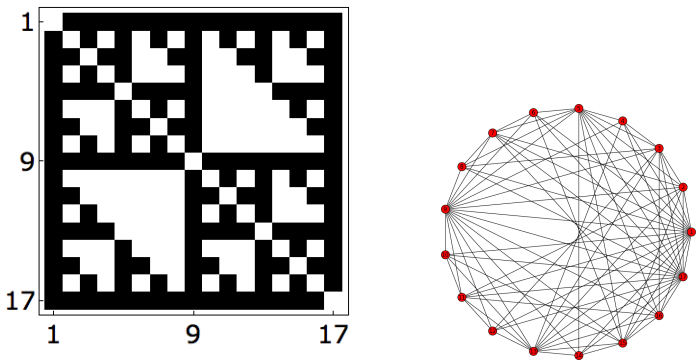
a) Initialization



b) Iteration 1



c) Iteration 2

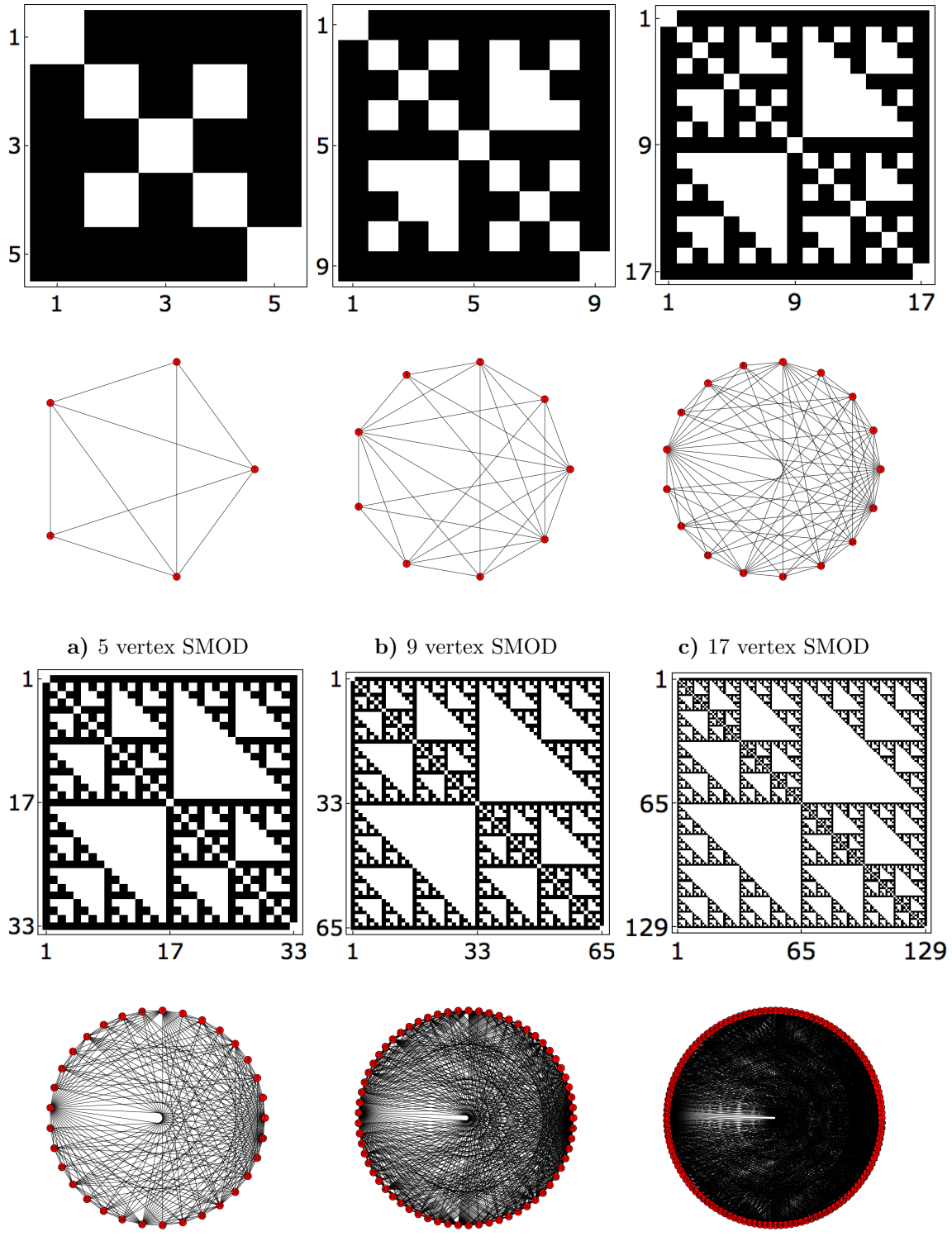


d) Final iteration 3

Figure 17. Steps of SMOD algorithm generating a graph with 17 vertices

5.3. Visualisations of SMOD graphs of small size

Figure 16 illustrates adjacency matrices generated by the SMOD algorithm and corresponding graphs representation for SMOD graphs with $2 \leq m \leq 7$.



a) 5 vertex SMOD **b)** 9 vertex SMOD **c)** 17 vertex SMOD

d) 33 vertex SMOD **e)** 65 vertex SMOD **f)** 129 vertex SMOD

Figure 18. SMOD graphs with $2 \leq m \leq 7$

6. Properties of SMOD graphs

This section outlines basic properties of SMOD graphs such as the relationship between order and the size of SMOD graphs, their diameter and mean path length as well as distance and vertex degree distributions. Most of these properties can be easily expressed analytically. We also look into the spectra of SMOD graphs and attempt to understand what information can be extracted from them.

6.1. Size of SMOD graphs

Theorem 2. If $n = 2^m + 1$ is the order of a SMOD graph, then its size is given by

$$|L| = 3^m = 3^{\log_2(n-1)} \tag{20}$$

Proof. Counting the edges removed at each step of the SMOD algorithm one gets the following formula

$$\begin{aligned} |L| &= \frac{1}{2}\left[n(n-1) - \sum_{k=1}^{m-1} 3^{k-1}\left(\frac{n-1}{2^k} - 1\right)\left(\frac{n-1}{2^k}\right)\right] \\ &= \frac{1}{2}\left[(2^m+1)2^m - \sum_{k=1}^{m-1} 3^{k-1}\left(\frac{2^m}{2^k} - 1\right)\left(\frac{2^m}{2^k}\right)\right] \end{aligned} \tag{21}$$

By mathematical induction let's show that

$$\frac{1}{2}\left[(2^m+1)2^m - \sum_{k=1}^{m-1} 3^{k-1}\left(\frac{2^m}{2^k} - 1\right)\left(\frac{2^m}{2^k}\right)\right] = 3^m \tag{22}$$

The simplest SMOD graph has order $n = 5$, which corresponds to $m = 2$. The formula holds for such m as

$$\frac{1}{2}\big[(2^2+1)2^2 - 2\big] = \frac{1}{2}(20-2) = 9 = 3^2 \tag{23}$$

Let's assume that the formula holds for some $m > 2$. We shall show that it also holds for $m+1$. Before we proceed, let's note that

$$\begin{aligned} &(2^{m+1}+1)2^{m+1} = 2\big[1+2+3+\cdots+2^{m+1}\big] \\ &= 2\big[1+2+3+\cdots+2^m+(2^m+1)+\cdots+(2^m+2^m)\big] \\ &= 2\big[\sum_{k=1}^{2^m} k + 2^m 2^m + \sum_{k=1}^{2^m} k\big] = 2\big[2\sum_{k=1}^{2^m} k + 4^m\big] \\ &= 2(2^m+1)2^m + 2\cdot 4^m \end{aligned} \tag{24}$$

and that

$$\begin{aligned} &\left(\frac{2^{m+1}}{2^k} - 1\right)\left(\frac{2^{m+1}}{2^k}\right) = \big(2^{m-k+1}-1\big)2^{m-k+1} \\ &= 2\big[1+2+3+\cdots+(2^{m-k}-1)+2^{m-k}+(2^{m-k}+1)+\cdots+(2^{m-k}+2^{m-k}-1)\big] \\ &= 2\big[\sum_{k=1}^{2^{m-k}-1} k + 2^{m-k}2^{m-k} + \sum_{k=1}^{2^{m-k}-1} k\big] = 2(2^{m-k}-1)2^{m-k} + 2\cdot 4^{m-k} \end{aligned} \tag{25}$$

Applying the inductive step and using (23) and (24) we get

$$\begin{aligned}
&\frac{1}{2}\left[(2^{m+1}+1)2^{m+1} - \sum_{k=1}^{m} 3^{k-1}\left(\frac{2^{m+1}}{2^k}-1\right)\left(\frac{2^{m+1}}{2^k}\right)\right] \\
&= \frac{1}{2}\left[2(2^m+1)2^m + 2\cdot 4^m - 2\sum_{k=1}^{m} 3^{k-1}[(2^{m-k}-1)2^{m-k} + 4^{m-k}]\right] \\
&= (2^m+1)2^m - \sum_{k=1}^{m-1} 3^{k-1}(2^{m-k}-1)2^{m-k} + 4^m - \frac{4^m}{3}\sum_{k=1}^{m-1}\left(\frac{3}{4}\right)^k \\
&= 2\cdot 3^m + 4^m - \frac{4^m}{3}\sum_{k=1}^{m-1}\left(\frac{3}{4}\right)^k = 3^{m+1} - 3^m + 4^m - \frac{4^m}{3}\sum_{k=1}^{m-1}\left(\frac{3}{4}\right)^k \\
&= 3^{m+1} - 3^m + 4^m - \frac{4^m}{3}\left(\frac{3/4 - (3/4)^{m+1}}{1-3/4}\right) \\
&= 3^{m+1} - 3^m + 4^m - \frac{4^m}{3}\left(3 - \frac{3^{m+1}}{4^m}\right) = 3^{m+1} - 3^m + 4^m - 4^m + 3^m \\
&= 3^{m+1}
\end{aligned} \tag{26}$$

By the induction assumption the formula holds for $m+1$, hence it holds for any $m \geq 2$. □

The figure below shows how the size of SMOD graphs changes as a function of m, where $n = 2^m + 1$ as n grows from 5 to 1,025 (or $2 \leq m \leq 10$)

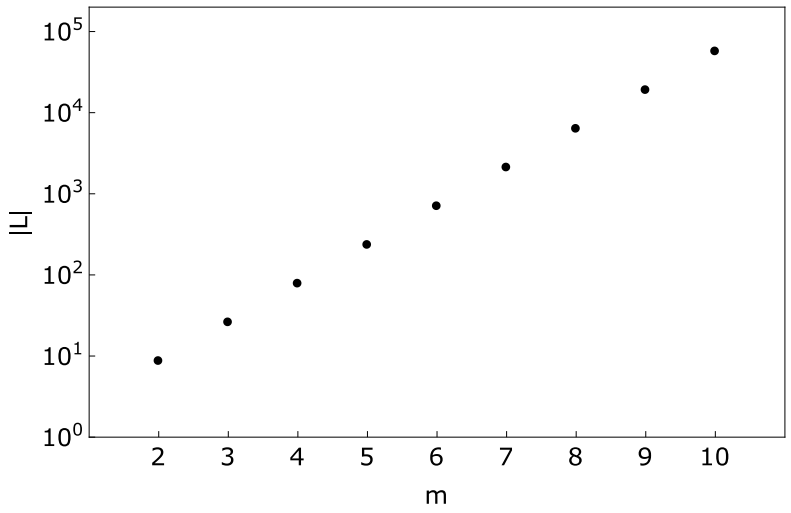


Figure 19. Sizes of SMOD graph with $2 \leq m \leq 10$

6.2. Diameter, mean path length and distances

Every SMOD graph of order n contains three S_n (n-star) subgraphs. Those three subgraphs are seen in the SMOD graphs' adjacency matrices as fully filled (except of the diagonal element) rows (or columns) 1, $\frac{n-1}{2}$ and n. This trivially corresponds to vertices that are connected to all other vertices. The S_n subgraphs are the minimum spanning trees of SMOD graphs. Since their diameter is 2 this implies that every SMOD graph has diameter 2.

An interesting conclusion follows from the above observation:

Theorem 3. There are no mutually orthogonal vectors in the column (or row) space of an adjacency matrix of a SMOD graph.

Proof. An SMOD graph of order n has exactly three S_n n-star subgraphs, which are also the minimum spanning trees. This means that the diameter of an SMOD graph of order n is 2 and thus the distance is at most 2, hence there always exists a walk of length 2 between any two vertices i and j. By Lemma 3 in [25] for any undirected graph the number of walks of length k between vertices i and j is equal $(A^k)_{ij}$, thus, since for all pairs of vertices i and j in SMOD graph there is always a walk corresponding to a distance ≤ 2 hence $(A^2)_{ij} = (A^T A)_{ij} > 0$. Therefore the scalar product of any two columns of an adjacency matrix of an SMOD is non-zero, and thus no pair of vectors from column space is orthogonal. □

The following Figure depicts how fast the mean path length approaches the diameter. This means that as the order of the graph grows, so does the number of pairs of vertices that are 2 hops away with respect those that are directly connected.

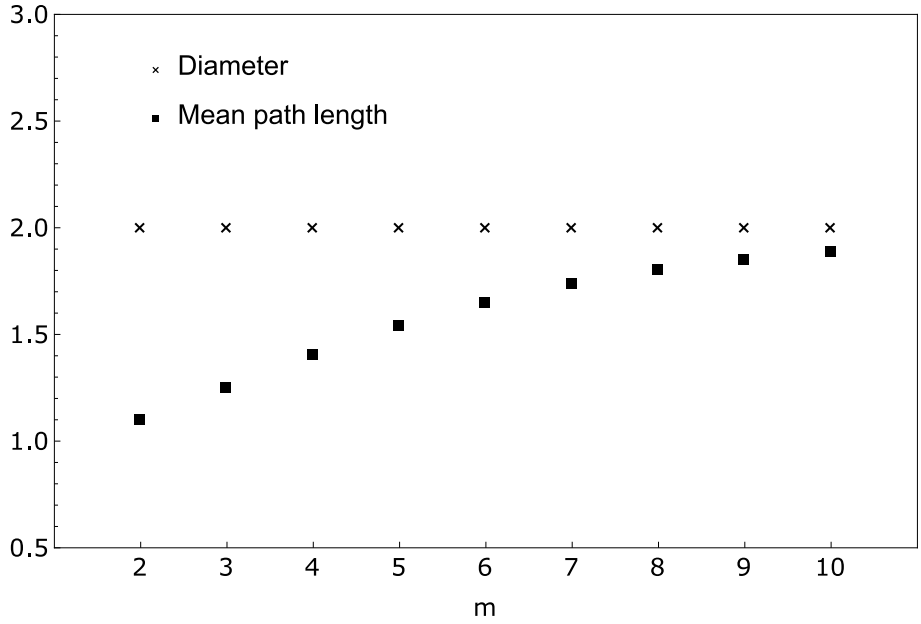


Figure 20. Graph diameters and mean path lengths of SMOD graphs

Since the diameter of any SMOD graph is 2, the distance between any two vertices can be either 1 or 2. This means that if two vertices i and j are not connected, then they are 2 hops away. In terms of the distance matrix, this says that wherever there is an off-diagonal 0 in the adjacency matrix there is 2 in the distance matrix. By this argument we see that the following formula holds for the distance matrix.

$$D = 2J - 2I - A \tag{27}$$

where J is all-ones matrix.

Using the argument leading to Eq. 26 and Theorem 2 one sees that the number of pairs of vertices, the distance between which is 2 is expressed by the following formula

$$\sum_{i \neq j} d(v_i, v_j)\Big|_2 = \frac{1}{2}(2^m + 1)2^m - 3^m = \frac{1}{2}n(n-1) - 3^{\log_2(n-1)} \tag{28}$$

The following figure shows the distance distribution in SMOD graphs.

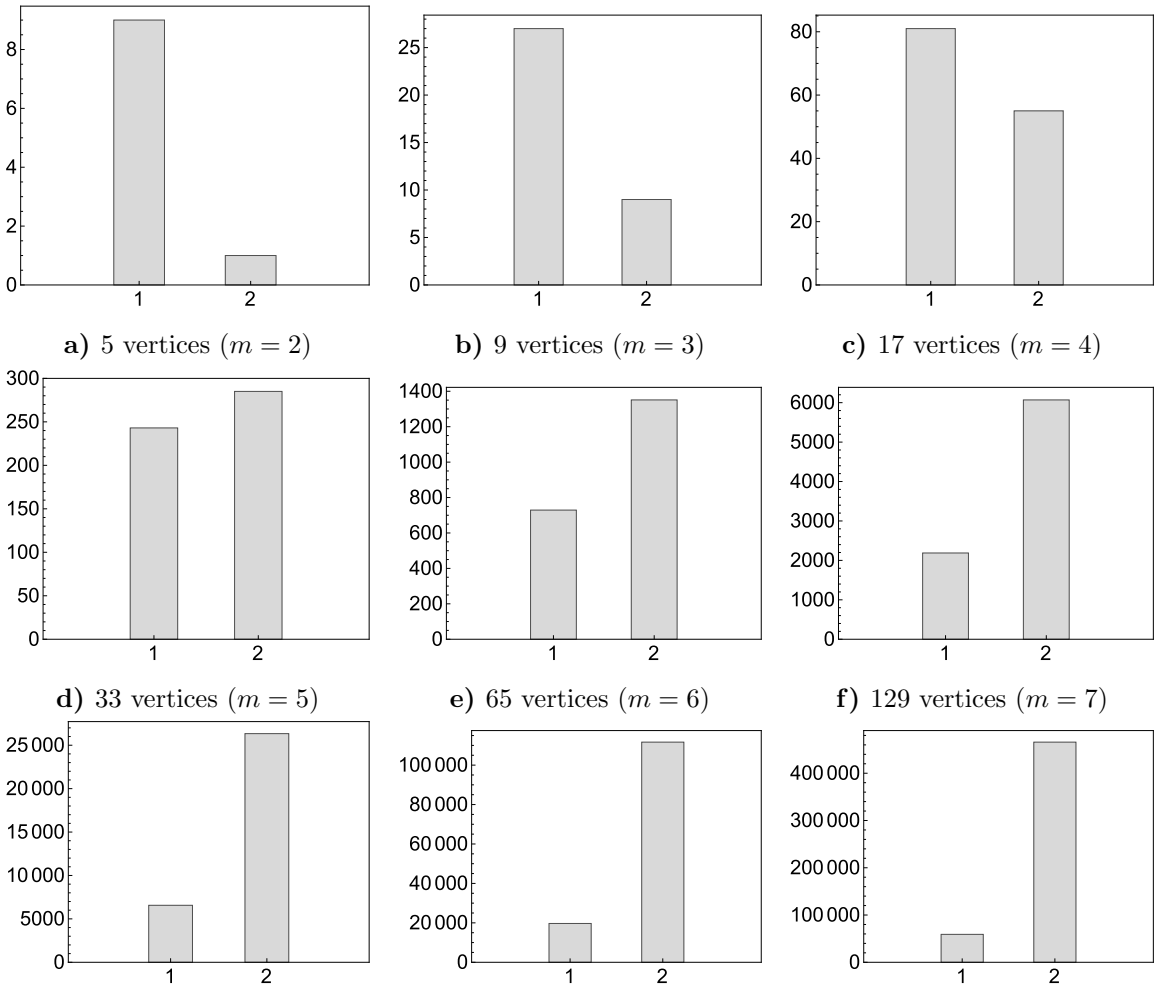
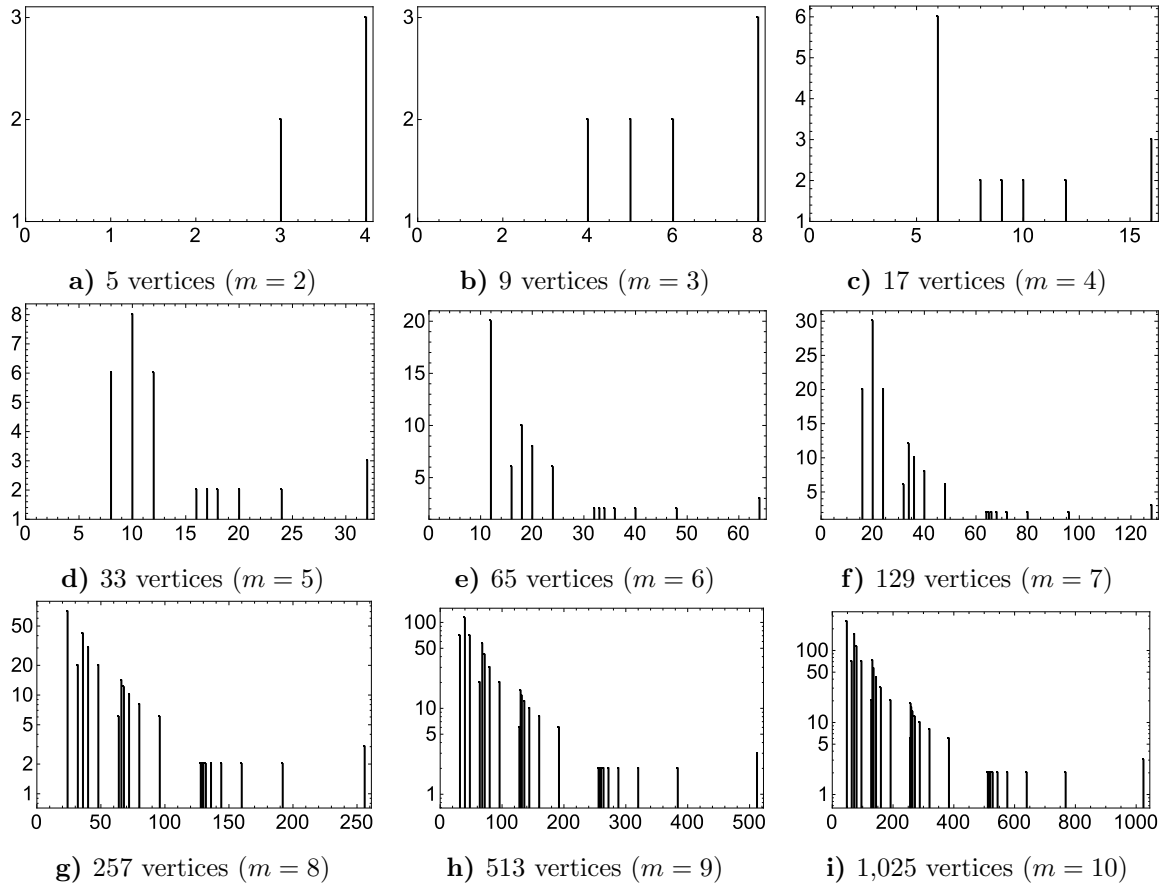


Figure 21. Distribution of path lengths in SMOD graphs with $2 \leq m \leq 10$

The figure depicts what can also be seen from Eq. 27: for $m \geq 5$ the pairs of vertices 2 hops away outnumber directly connected pairs.

6.3. Vertex degrees and their distribution

In general, a low diameter of a graph implies existence of vertices of high degree. This is the case with SMOD graphs, which have diameter 2 regardless of the graph order, but at the same time three vertices have degree $n - 1 = 2^m$ and large number of vertices are of relatively high degree.



a) 5 vertices ($m = 2$) **b)** 9 vertices ($m = 3$) **c)** 17 vertices ($m = 4$)

d) 33 vertices ($m = 5$) **e)** 65 vertices ($m = 6$) **f)** 129 vertices ($m = 7$)

g) 257 vertices ($m = 8$) **h)** 513 vertices ($m = 9$) **i)** 1,025 vertices ($m = 10$)

Figure 22. Vertex degree *vs.* number of vertices of that degree in SMOD graphs with $2 \leq m \leq 10$

6.4. Eigenvalues of SMOD graphs

Here again, we denote the greatest eigenvalue by λ_1 and the least eigenvalue by λ_n. As with MOD graphs, SMOD graphs have the the greatest eigenvalue equal to the spectral radius, that is $\lambda_1 = \rho(A(G))$.

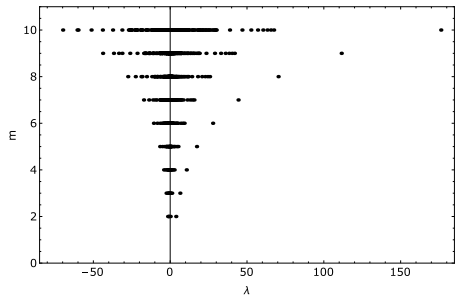


Figure 23. Distribution of eigenvalues for SMOD graphs with $2 \leq m \leq 10$.

Figure 21 above represents the distribution of eigenvalues for SMOD graphs of order $n = 2^m + 1$ for $2 \leq m \leq 10$. There is a noticeable gap between λ_1 and λ_2. Detailed study of exact interpretation of the SMOD graph eigenvalues is outside the scope of this paper.

6.5. Edge ratio of SMOD graphs

Regardless of the order, SMOD graphs have diameter 2. From Eqs. 19 and 27 and the definition of the mean path length of an undirected graph it follows that the mean path length of a SMOD graph approaches 2 for large m. It is also seen in Figure 18. A question we raise here is how does SMOD graph size is compared with complete graph size. First, we compute sizes of all graphs of order $n = 2^m + 1$, for $2 \leq m \leq 10$. The results are depicted in the figure below.

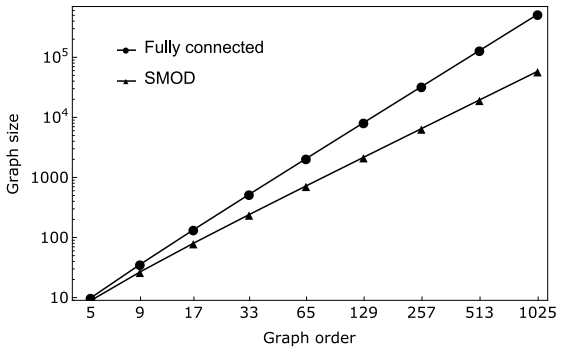


Figure 24. Comparison of graph sizes for SMOD graphs and complete graphs

Figure 23 below illustrates edge ratio of SMOD graphs which we define as a size ratio of a SMOD graph to a complete graph. The tick marks in the figure correspond to consecutive values of m, where $n = 2^m + 1$. For $m = 2$, the ratio is 0.9 and it drops to 0.1 for $m = 10$. The edge ratio converges exponentially to 0 as m grows large.

This is expressed analytically in the following manner

$$\lim_{m\to\infty} L_{ratio} = \lim_{m\to\infty} \frac{3^m}{\frac{1}{2}(2^m+1)2^m} = 0 \tag{29}$$

Table 3 on the following page summarises the order, size, diameter and mean path length of SMOD graphs for $2 \leq m \leq 10$.

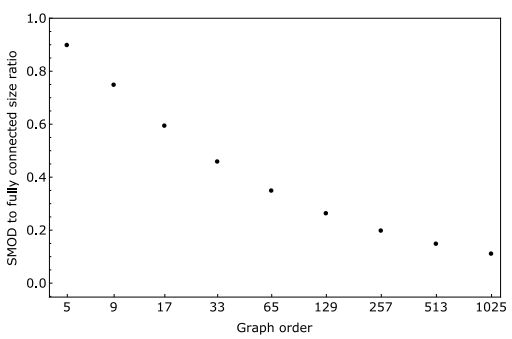


Figure 25. Size ratio Comparison of SMOD and fully connected graphs

Table 3. Comparison of SMOD, SlimFly [14] and the complete graphs. SlimFly is compared wherever it can attain similar order. See Table 2 for comparison of SlimFly with other graphs

Graph	Order	Size	Diameter	Mean path length	Bisection bandwidth
SMOD	**5**	**9**	**2**	**1.10**	5
Complete	5	10	1	1.00	6
SMOD	**9**	**27**	**2**	**1.25**	14
Complete	9	36	1	1.00	20
SMOD	**17**	**81**	**2**	**1.40**	41
SlimFly	18	45	2	1.70	15
Complete	17	136	1	1.00	72
SMOD	**33**	**243**	**2**	**1.54**	130
Complete	33	528	1	1.00	272
SMOD	**65**	**729**	**2**	**1.64**	276
SlimFly	50	539	2	1.86	65
Complete	65	2,080	1	1.00	1,056
SMOD	**129**	**2,187**	**2**	**1.74**	794
Complete	129	8,256	1	1.00	4,158
SMOD	**257**	**6,561**	**2**	**1.80**	2,317
SlimFly	242	2,057	2	1.93	671
Complete	257	32,896	1	1.00	16,500
SMOD	**513**	**19,683**	**2**	**1.85**	7,068
Complete	513	131,328	1	1.00	65,736
SMOD	**1,025**	**59,049**	**2**	**1.89**	20,262
SlimFly	1058	18,515	2	1.97	6,095
Complete	1,025	524,800	1	1.00	262,446

Conclusions

We present a novel method and algorithms for generating new classes of MOD and SMOD graphs. We stress that our primary objective in the present work is to propose fast and effective algorithms for creating sparse adjacency matrices by iterative removal of entire groups of edges. This translates to creating well connected graphs of arbitrarily large order but of reasonably small size and also having other properties that render some of them to be interesting candidates for supercomputer interconnect topologies. However, we do not claim that this process will lead to "optimal" interconnect network topologies - irrespective of the definition of "optimal". The central object for this method is the adjacency matrix, and we derive the graphs, and all their properties from this matrix. The starting graph is always the complete graph. We construct graphs of arbitrary order $n = 2^m$ or $n = 2^m + 1$, for MOD and SMOD graphs, respectively, and gradually, by algorithmic removal of edges, arrive at the final graph. We also propose a halted

MOD algorithm which leads to graphs $aMOD(m,c)$ with cliques of order 2^{m-c}. We compare the main graph properties such as diameter, mean path length, histogram of distances and size to all typical graph topologies represented in the most important supercomputers that are currently on the market. It turns out that our MOD graphs are better than almost all of these supercomputer interconnect topologies, with the exception of dragonfly topology, for graphs of some large orders.

The algorithm was designed to introduce a systematic and simple way of removing as large number of edges as possible at each iterative step, starting from the complete graph, and at the same time to preserve good graph connectivity, and arrive at final graphs of reasonably small size. The method is fast, almost all important graph properties can easily be derived from the combinatorics of the algorithm.

Clearly, besides the two algorithms described here, there may be great number of similar schemes leading to other graphs of similarly interesting, and perhaps even "better", properties.

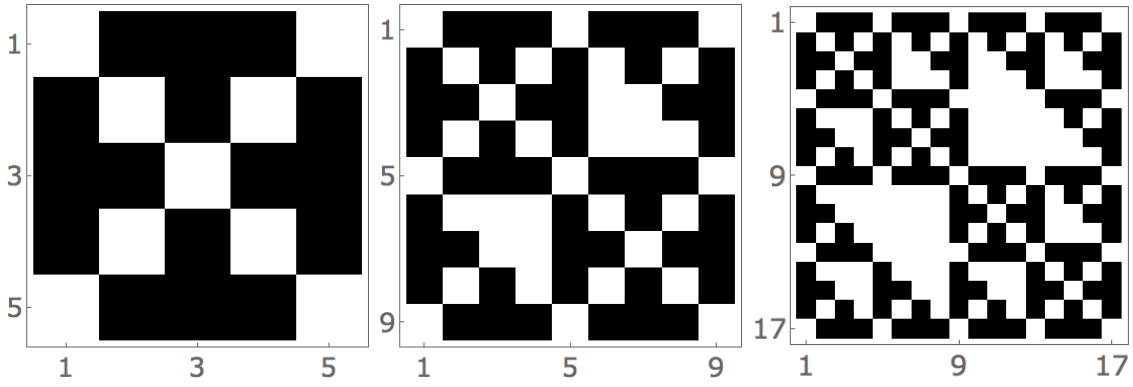
Here are some examples of the other schemes that have not been investigated here. We plan to investigate these alternatives and report detailed results in the future.

1. A trivial modification of MOD algorithm described in Fig. 1 would be to substitute P_e at the first iteration step with the symmetric matrix $P_s \in \mathbb{R}^{n/2 \times n/2}$ defined as

$$(P_s)_{ij} = \begin{cases} 1 & i = n/2, j = 1 \\ 1 & i = 1, j = n/2 \\ 0 & \text{otherwise} \end{cases}$$

this will lead to a *regular* graph *i.e.* a graph with all vertices of the same degree. Eq. 6 for such a variant of MOD graph would read: $Au = (m+1, m+1, ..., m+1, m+1)^T$. This graph will have a size bigger by 1 than our MOD graph, *i.e.* $|L| = \frac{n}{2}(log_2 n + 1) = (m+1)2^{m-1}$.

2. Modification of SMOD algorithm to remove 3^{p-1} extra edges - "the centres of star graphs" at each p^{th} iteration. This is illustrated below in Fig. 24 (a)-(c) and should be compared with Fig. 16 (a)-(c). Three most important implications of such a new augmented algorithm are: i) this will also lead to graphs of diameter 2, ii) the graphs will have smaller size, iii) all vertices with of the highest order will be removed (see Fig. 20 (a)-(i)). All these characteristics make such modified SMOD graphs better candidates for interconnects in Big Data era (up to some reasonable size $n = 2^m + 1$).



a) 5 vertex modified SMOD **b)** 9 vertex modified SMOD **c)** 17 vertex modified SMOD

Figure 26. Adjacency matrices of modified SMOD graphs with lowered maximum vertex degrees

3. Combination of MOD and SMOD algorithms. This process will start with several steps performed with one of the algorithms, and than switch to the other one. Clearly this will lead to graphs with "off-diagonal" parts further away from the main diagonal akin to the first graph, with all blocks adjacent to the main diagonal having a character of the second graph family. It is also clear that such combined graphs are also *embedded* graphs of one in to the other graph class.

A cursory glance at Figs. 3, 16 and 24 leads us to an immediate conclusion that each MOD, SMOD or modified SMOD graph is composed of *self-similar* graphs of all smaller orders. Again, this can be cast in the statement that these graph families are *embeddings* of graphs of smaller order. The most remarkable property of the SMOD graphs is that irrespective of the graph order, the diameter is constant and equals 2. However, the size of the graph, or the total number of edges, is about 10% of the size of a complete graph of the same order and there are vertices of rather high order. These characteristics may prove useful in considering SMOD graphs as possible candidates for interconnect topologies for Big Data computer architectures, provided the vertex order does not exceed technological capabilities, *i.e.* the number of ports in a typical switch. One may however speculate that if such topology is implemented as printed circuit paths or in optical switch - the high radix may not be a serious obstacle anymore, considering substantial reduction in the number of required links. It would be also interesting to investigate further how well MOD and SMOD graphs topologies match communication patterns in typical computational algorithms. We plan to devote a separate study to this open problem.

References

1. `http://www.infoworld.com/d/computer-hardware/china-building-100-petaflop-supercomputer-206072`. Last accessed: 2014.08.05.
2. `http://www.top500.org/lists/2014/06/`. Last accessed: 2015.10.05.
3. `http://www.cray.com/Products/Computing/XK7/Technology.aspx`. Last accessed: 2014.08.05.
4. `https://bluewaters.ncsa.illinois.edu/user-guide`. Last accessed: 2014.08.05.
5. `https://computing.llnl.gov/tutorials/bgq/#Networks`. Last accessed: 2014.08.05.
6. `http://www.nas.nasa.gov/hecc/resources/pleiades.html`. Last accessed: 2014.08.05.
7. `http://www.cscs.ch/computers/pizdaint/index.html`. Last accessed: 2014.08.05.
8. `http://www.top500.org/system/178515`. Last accessed: 2015.10.05.
9. Report on the workshop on extreme-scale solvers: Transitions to future architectures. Technical report, Office of Advanced Scientific Computing Research, U.S. Department of Energy, March 2012. `http://science.energy.gov/ascr/news-and-resources/program-documents/`, Last accessed: 2014.08.05.
10. On advanced scientific computing research exascale mathematics working group (emwg) report for 2014. Technical report, The Department of Energy (DOE) Office of Science Pro-

gram, 2014. https://collab.mcs.anl.gov/display/examath/Exascale+Mathematics+Home, Last accessed: 2014.08.05.

11. N. R. Adiga et al. BlueGene/L torus interconnection network. *IBM Journal of Research and Development*, 45(2.3):265–276, 2005.
12. Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 11(42):36–40, 2009.
13. Richard F Barrett, Courtenay T Vaughan, Simon D Hammond, and D Roweth. Application explorations for future interconnects. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1717–1724. IEEE, 2013.
14. Maciej Besta and Torsten Hoefler. Slim fly: a cost effective low-diameter network topology. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 348–359. IEEE, 2014.
15. RB Borie, R Gary Parker, and CA Tovey. Recursively constructed graphs. *Handbook of Graph Theory*, pages 99–125, 2004.
16. J. P. Brown. Hypercubes. *Practical Computing, Vol. 5 No. 4*, pages 97–99, 1982.
17. Dragoš M Cvetković, Peter Rowlinson, and Slobodan Simic. *Eigenspaces of graphs*. Number 66. Cambridge University Press, 1997.
18. William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.
19. Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray cascade: a scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 103. IEEE Computer Society Press, 2012.
20. M. Kan. China is building a 100-petaflop supercomputer. *IDG News Service, infoworld.com (2012-10-31)*.
21. John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 77–88. IEEE Computer Society, 2008.
22. F Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes*. Elsevier, 2014.
23. Matt Knepley Mark F. Adams, Jed Brown. Low-communication techniques for extreme-scale multilevel solvers. https://collab.mcs.anl.gov/display/examath/Submitted+Papers. Last accessed: 2014.08.05.
24. Richard Otter. The number of trees. *Annals of Mathematics*, pages 583–599, 1948.
25. Piet Van Mieghem. *Graph spectra for complex networks*. Cambridge University Press, 2010.

DOI: 10.14529/jsfi150403

Multi-Scale Supercomputing of Large Molecular Aggregates: A Case Study of the Light-Harvesting Photosynthetic Center

Igor V. Polyakov [1], ***Alexander A. Moskovsky*** [1, 2, 3], ***Alexander V. Nemukhin*** [1, 3]

Numerical solution of the quantum mechanical Schrdinger equation is required to model electronic excitations in the light-harvesting photosynthetic complexes composed of up to millions of atoms. We demonstrate that the modern supercomputers can be used to treat electronic structure calculations in large molecular aggregates if proper multi-scale massive-parallel approaches are applied. We show that three-level parallelization scheme based on the novel numerical algorithms assuming fragmentation of a light-harvesting complex allows us to considerably reduce the high scaling of *ab initio* quantum chemistry methods. More specifically, we have applied the time-dependent density functional theory based on the fragment molecular orbital presentation (FMO-TDDFT) implemented in modern supercomputers to obtain realistic estimation of the electronic excitation in the complex. The application shows a good overall scaling.

Keywords: quantum chemistry, fragmentation, multi-scale approaches, parallel algorithms.

Introduction

Application of *ab initio* quantum-chemical algorithms aiming at numerical solutions of the Schrdinger equation for molecules has served as a polygon for testing high-performance computers for a long period of time. One of the reasons is that the computational cost of such simulations increases dramatically with the system size N, namely, up to N^6, especially if electronically excited states are involved. Presently, quantum-chemical calculations can be performed efficiently through a number of modern software readily available for any of supercomputer architectures. Most of the popular programs are fairly scalable and perform well with the big x86-64 clusters, some of these applications are capable to harness accelerators such as NVIDIA K40 or Xeon Phi devices. Certain algorithms also require vast amounts of memory per core which can be a problem for most computational clusters. That is why the routine calculations are usually conducted for moderate sized systems containing several hundreds of atoms in total. They usually scale well for up to 256 cores on the Infiniband clusters.

In this respect characterization of the photosynthetic molecular machines responsible for conversion of solar energy to chemical energy presents one of the greatest challenges for quantum-based simulations. Relevant model systems mimicking the reaction centers (RC) and the light harvesting (LH) complexes should incorporate up to millions of interacting atoms, and calculations of electronic excitations in the photosensitive part of such complexes constitute the most difficult problem.

In attempts to reduce the high scaling of *ab initio* methods and novel numerical algorithms based on fragmentation of a large molecular aggregate have begun to extend the traditional quantum-chemistry approaches. Recently developed methodology of fragment molecular orbitals (FMO) [2] [6] [7] opens the way for application of algorithms of quantum chemistry for a very

[1]Department of Chemistry, M.V. Lomonosov Moscow State University, Leninskie Gory 1/3, Moscow 119991, Russian Federation
[2]ZAO RSC Technologies, Kutuzovskiy av., 36/23, Moscow 121170, Russian Federation
[3]N.M. Emanuel Institute of Biochemical Physics, Russian Academy of Sciences, Kosygina 4, Moscow 119334, Russian Federation

large molecular system assuming a properly designed fragmentation scheme. We show here how this method can be applied for a photosynthetic LH-RC complex comprising a half of million atoms. Presumably, the FMO technique belongs to the group of highly parallel methods of quantum chemistry. In this work we check the corresponding options using current implementation of the FMO algorithms in modern facilities utilizing both large and small computational clusters.

1. Models and Methods

It is beyond the scope of this paper to describe construction of a model molecular system for the photosynthetic center. We only mention that such work presents a separate highly demanding computational problem and requires applications of consuming molecular mechanics and molecular dynamics calculations. We refer to our recent work [5] in which the bacterial photosynthetic core complex LH-RC composed of a half of million atoms has been modeled. Here we use the atomic coordinates of this all-atom three-dimensional model system to characterize its electronic excitations.

An elliptical ring of 32 bacteriochlorophyll (BChl) chromophores (each of them being a very large molecule) embedded in the LH polypeptide chains lies in the heart of molecular machinery. Subtle interaction of the BChl molecules between themselves as well as with the surrounding molecular groups plays an essential role in the excitation process. It is unrealistic to expect that a direct calculation of the electronic excitations in such system by using *ab initio* quantum chemistry methods will be carried out in the nearest future. Application of the multi-scale fragmentation technique [2] [7] is an important step toward the quantum-based characterization of the system.

According to the fragment molecular orbital (FMO) method a molecular system is divided into a number of fragments (N). The quantum Hamiltonian of the fragment I is written in the conventional form [6] (Eq.1)

$$H_I = \sum_i^{n_I}\{-\frac{1}{2}\bar{V}_i^2 - \sum_s^{all-atoms}\frac{Z_s}{|r_i - r_s|} + \sum_{J\neq I}^{N}\int dr'\frac{\rho_J(r')}{|r_i - r'|}\} + \sum_{i>j}^{n_I}\frac{1}{|r_i - r_j|} + \sum_{s>t}^{fragment}\frac{Z_s Z_t}{|r_s - r_t|} \quad (1)$$

Here n_I - number of electrons in the fragment I, Z_s - nuclear charge of atom s, $\rho_J(r)$- electron density of fragment J. After solving the Schrdinger equations at an appropriate level by using either the Hartree-Fock (HF) or the density functional theory (DFT) one obtains the fragment energies E_I and fragment pair (in the second-order FMO theory, FMO2) energies E_{IJ}. The total energy of the system can thus be expressed as follows [1] (Eq. 2)

$$E = \sum_{I=1}^{N} E_I + \sum_{I=1}^{N}(E_{IJ} - E_I - E_J) \quad (2)$$

To treat the electronic excitations the time-dependent density functional theory (TDDFT) based upon the fragment molecular orbital presentation [1] is applied. In the FMO-TDDFT method the excitation is assumed to be localized mostly on one fragment (M), other fragments contribute the additive two-body corrections (Eq. 3)

$$E^* = E_M^* + \sum_{I\neq M} E_I^0 + \sum_{I\neq M}(E_{MI}^* - E_M^* - E_I^0) + \sum_{\substack{I>J \\ I,J\neq M}}(E_{IJ}^0 - E_M^* - E_I^0) \quad (3)$$

where E^0 notation is for the ground and E^* for the excited state energies. The excitation energy ω in the FMO2-TDDFT approach is given as a sum of the target monomer excitation energy ω_M and a sum of dimer corrections:

$$\omega \equiv E^* - E^0 = \omega_M + \sum_{I \neq M} (\omega_{MI} - \omega_M) \tag{4}$$

The FMO algorithms are implemented in modern versions of the GAMESS(US) quantum-chemistry package [10] which are currently running on Moscow State University Lomonosov supercomputers [9] and on MVS-10P cluster in Joint Supercomputer Center of the Russian Academy of Sciences. An essential feature of this software is the use of generalized distributed data interface (GDDI) [3] which allows the 2-level hierarchical parallelization scheme. This approach allows individual tasks to be assigned to groups that execute them nearly independently of each other approaching linear scalability at the group level. A commonly used parallel library MPI is also capable to divide nodes into groups, but it is a very basic and static interface. The GDDI supports distributed memory operations localized within groups, whereas distributed memory implementation in GAMESS cannot use the original MPI group-divided nodes.

After the initialization in a typical FMO job the single-point HF or DFT calculation (see Eq.(1)) is performed on each fragment separately in the mean Coulomb field of other monomers. This calculation is performed by one DDI group nearly independently of the others. One cycle of the single-point monomer self-consistent field runs generates a new density, thus changing the Coulomb field; these cycles are run until convergence is obtained. This yields the monomer densities that are then used during the dimer runs, also done independently by DDI groups. During this second step the dimers are constructed from each pair of monomers, and the initial density is taken to be the sum of the two monomer densities. Then, a self-consistent field (SCF) calculation is performed for each dimer.

Our previous experience to employ the FMO method to simulate properties of a biomolecular system [8] has resulted in the encouraging conclusions. Linear scalability up to 1536 cores on the RSC Tornado computational cluster has been shown and a limit of 5000 cores has been reported for the system of 2570 atoms.

In this work, FMO2-TDDFT calculations have been carried out for a model photosynthetic center using Lomonosov-2 supercomputer and MVS-10P cluster. GAMESS(US) version 5 (from December 2014 (R1)) has been compiled with Intel Fortran compiler version 14. Intel MKL BLAS library has been used in both cases; however, OpenMPI-1.8.4 has been used on Lomonosov-2 and Intel MPI-4.1 on MVS-10P.

2. Application

Even within the FMO construct it is unpractical to assign all 32 BChl chromophores to a single fragment, to directly compute its excitation energy, and to estimate the contributions from other fragments representing the environmental molecular groups (see Eq.(4)). Therefore, we rely on effective Hamiltonian technique [4] which is often applied to compute excitonic levels of light-harvesting complexes. Usually the empirical or semi-empirical formulae are employed within this approach [4] [12] but in this work we evaluate diagonal and near-diagonal elements of the entire 32 x 32 Hamiltonian matrix by the *ab initio* type quantum calculations. This requires to conduct 32 FMO2-TDDFT calculations which can be run independently, thus, adding a 3rd parallel level on top of regular GDDI 2-level hierarchical parallelization scheme of FMO2-

TDDFT [3]. The remaining off-diagonal elements are small and can be calculated through the formula for dipole-induced dipole coupling as given in Ref. [4].

More specifically, each of 32 calculations is centered around one particular BChl molecule from the BChl ring. This is a target fragment for the FMO-TDDFT calculation (see Eq.(3)). The corrections due to the environmental molecular groups computed within the FMO2-TDDFT method (Eq.(4)) are evaluated as follows. All atoms lying within a certain limit from the selected BChl (grouped to the corresponding fragments) are included into a model subsystem.

To illustrate this computational scheme in more details we refer to the model system of the photosynthetic center described in Ref. [5]. We have selected BChl No.1 as a target chromophore created the subsystem centered around BChl No.1 by cutting all molecules and residues lying outside the limit of 10 from BChl No.1. The subsystem size was 3624 atoms including those from a group of 5 BChl molecules (BChl No.31,32,1,2,3). These atoms were combined to 172 fragments for the FMO scheme; also 97 water molecules were added. All broken peptide bonds on the edges of the models were capped with hydrogen atoms. Protein chains were split as one amino acid residue per fragment. Carotenoid residues were split into four equal parts, membrane residues were split into six parts. BChl molecules were divided as follows - the ring with the magnesium atom was assigned to one fragment and the remaining parts to three other fragments. The histidines coordinating the magnesium ions were added to BChl ring fragments.

The two-layer model was employed in every FMO2-TDDFT calculation. Layer 1 was treated at the HF/3-21G level whereas layer 2 - at the more accurate DFT level CAM-B3LYP/6-31G*. The 2nd layer consisted of BChl molecules No.32,1,2 and of all fragments in a close contact with target BChl No.1 fragment.

The excitation energy of 1.658 eV for the central BChl No.1 fragment was obtained. The FMO2-TDDFT corrections (-0.107eV) shifted the resulting excitation energy to 1.551 eV. This value is very close to the results of the excitonic model [4] obtained with the empirically adjusted parameters for the effective Hamiltonian, 1.48 eV, thus, providing support to our computational scheme.

3. Technical aspects

This application shows good overall scaling of FMO algorithms as illustrated in Table 1. Calculation time (minutes) of the FMO-TDDFT algorithms attainable by Lomonosov-2 supercomputer.

It takes little time to achieve convergence for the 1^{st} layer consisting of 172 fragments when using the 3-21G basis set in HF calculations. Correspondingly, this part of the calculation does not benefit from a large number of cores per group (240 vs 480) but thrives from the increased number of groups (3 and 6 vs 1). The 2^{nd} layer consists of only 42 fragments but basic set (6-31G*) is considerably larger which means a greater computational effort per fragment, longer runtime and permits the better intergroup scaling. It should be noted the SCF calculations for monomers in both layers obviously benefit from the increased GDDI group count. The excitation energy calculation (Eq.(4)) for our system requires computations of several large (with 1592-877 basis functions) TD-CAM-B3LYP dimer independent tasks. It takes the greatest walltime ($\sim 70\%$), while the scaling is good - 81% for 480 vs 130 cores in a single group.

In this application the nodes were evenly assigned to each group which is an obvious drawback as can be seen from timing numbers for 2^{nd} layer SCF calculations for 480/3 and 480/6 tests. This is due to a low count of the computational tasks per FMO iteration (42). Also there

Table 1. Calculation time (minutes) of the FMO-TDDFT algorithms attainable by Lomonosov-2 supercomputer

Number of cores/groups	Layer1	Layer2	TDDFT	Total
130/1	54.6	210	730.7	995.3
240/1	37.1	127	423.2	587.3
240/3	28.1	109.1	403.6	540.8
480/1	28.2	82.4	243.6	354.5
480/3	16.9	58.9	211.5	287.3
480/6	14.4	89.7	209.5	313.6
720/6	10.4	61.9	142.9	215.2

are five very large BChl fragments that require large calculation efforts while other fragments are considerably smaller. However, GDDI implementation in GAMESS(US) is very flexible and allows one to use different number of groups for different kinds of computational tasks by varying the ngrfmo parameter. It also enables manual node division into groups using the mannod parameter. Correspondingly, computational scheme should benefit from employing ~ 6 groups with 512-128 cores per group. By using a total of $\sim$ 2000 thousands of CPU cores with almost linear scalability it should take less than 100 minutes to complete the calculation. One should note that GAMESS(US) TDDFT code does not use distributed memory algorithms. The replicated memory demands are quite high for large tasks; in our application, it took up to 2 Gigabytes per core.

As mentioned earlier, in order to fully describe the light-harvesting antenna using the effective Hamiltonian technique 32 independent calculations have to be conducted to obtain all diagonal and most important off-diagonal matrix elements. On the one hand, this scheme enables an efficient use of tens of thousands of processor cores simultaneously, and on the other hand it should still be operational with only several hundreds of cores used for a long period of time. This is a very important issue because large computational clusters are not often available for a user.

The constantly updated TOP500 list reflects innovations and modern tendencies in supercomputing. Remarkably, current No.1 (Tianhe-2) and 2 (Titan) positions are held by supercomputers using Intel Xeon Phi and NVIDIA K20x. Innovations in quantum chemistry software driven by modern supercomputers design enable efficient use of accelerators [11] to carry out the most computationally demanding and complex tasks in molecular modeling. Although our computations are capable to take advantages of large computational clusters with x86-64 processors currently attainable FMO calculations with GAMESS(US) software are unable to efficiently utilize either NVIDIA CUDA or Xeon Phi devices.

Comparing Lomonosov-2 configuration with that of MVS-10P we see that nodes on Lomonosov can provide only 10 CPU cores to a generic application. RAM bandwidth is not saturated by 10 cores (Intel Xeon E5V2, IvyBridge) and better application performance (per node) can be expected on MVS-10P nodes where two sockets provide together 16 cores of Intel Xeon E5 (Sandybridge) on close clock frequency. However the number of nodes is $\sim$ 200 for MVS-10P and $\sim$ 1000 for Lomonosov-2. Both computational clusters use Mellanox FDR Infiniband. MVS-10P nodes have local SSD storage that can be useful in a number of quantum

chemistry methods for temporary storage (ERI data). Our application is not demanding to the parallel file storage, so in this respect the system differences are of minor importance.

Conclusion

This work demonstrates efficiency of novel quantum-based computational algorithms implemented in the modern supercomputers to compute the most difficult properties of large molecular aggregates such as electronic excitations in the photosynthetic light-harvesting complexes. A notable feature of these calculations is a use of the three-level parallelization scheme which allows one to gain advantages of the proper fragmentation of a model system. A good overall scaling of the fragment molecular orbital algorithms is demonstrated.

This work is supported by the Program No.24 of the Presidium of the Russian Academy of Sciences.

References

1. M. Chiba, D. Fedorov, and K. Kitaura. Time-dependent density functional theory based upon the fragment molecular orbital method. *J. Chem. Phys.*, 27, 2007.

2. D. Fedorov, K. Kitaura, E. Starikov, J. Lewis, and S. Tanaka. *Modern Methods for Theoretical Physical Chemistry of Biopolymers*. September 2006.

3. D. Fedorov, R. Olson, K. Kitaura, M. Gordon, and S. Koseki. A new hierarchical parallelization scheme: generalized distributed data interface (gddi), and an application to the fragment molecular orbital method (fmo). *J. Comput. Chem.*, 25:872—880, 2004.

4. X. Hu, T. Ritz, A. Damjanovic, and K. Schulten. Pigment organization and transfer of electronic excitation in the photosynthetic unit of purple bacteria. *J. Phys. Chem.*, 101:3854—3871, 1997.

5. M. Khrenova, A. Nemukhin, B. Grigorenko, P. Wang, and J.-P. Zhang. All-atom structures and calcium binding sites of the bacterial photosynthetic lh1-rc core complex from thermochromatium tepidum. *J. Mol. Model.*, 20:1–6, 2014.

6. K. Kitaura, E. Ikeo, T. Asada, T. Nakano, and M. Uebayasi. *Fragment molecular orbital method: an approximate computational method for large molecules*. November 1999.

7. T. Nagata, D. Fedorov, and K. Kitaura. Linear-scaling techniques in computational chemistry and physics. *Springer*, 13:17–64, 2011.

8. I. Polyakov, L. Grigorenko, A. Moskovsky, Vl. Pentkovski, and A. Nemukhin. Towards quantum-based modeling of enzymatic reaction pathways: Application to the acetylholinesterase catalysis. *Chem. Phys. Lett.*, 556:251—255, 2013.

9. V. Sadovnichy, A. Tikhonravov, Vl Voevodin, and V. Opanasenko. Lomonosov: Supercomputing at moscow state university. In *Contemporary High Performance Computing: From*

Petascale toward Exascale, Chapman and Hall/CRC Computational Science, pages 283–307, Boca Raton, United States, 2013. Boca Raton, United States.

10. M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, and J. Montgomery. *General Atomic and Molecular Electronic Structure System.*, volume 14. 1993.

11. B. Scott and B. Levine. Nanoscale multireference quantum chemistry: Full configuration interaction on graphical processing units. *J. Chem. Theory Comput.*, 11:4708—4716, 2015.

12. S. Tretiak, C. Middleton, V. Chernyak, and S. Mukamel. Bacteriochlorophyll and carotenoid excitonic couplings in the lh2 system of purple bacteria. *J. Phys. Chem.*, 104:9540—9553, 2000.

DOI: 10.14529/jsfi150404

INMOST Parallel Platform: Framework for Numerical Modeling

Alexander A. Danilov[1], ***Kirill M. Terekhov***[1,2], ***Igor N. Konshin***[1], ***Yuri V. Vassilevski***[1]

INMOST program platform allows a user to work with distributed data on general meshes. Description of the platform structure, interrelation of mesh elements, work with ghost cells, distribution and redistribution of mesh data is presented, as well as special aspects of the program platform implementation and usage. This paper aims to cover the following research topics: efficient distributed unstructured mesh representation data structure, flexible templates for numerical schemes implementation, convenient framework for parallel linear systems assesment and solution. For one of the specific tasks the exploiting of INMOST program platform is demonstrated on all stages of numerical modeling: distributed meshes construction, attachment of data to mesh elements, use of mesh data for problem discretization, as well as parallel solution of resulting linear systems. INMOST is a newly developed, flexible, and efficient numerical analysis framework that provides scientists with the infrastructure for creating highly scalable high performance computing modeling applications.

Keywords: distributed mesh, polyhedral mesh, parallel framework, numerical modeling.

Introduction

The amount of software for unstructured mesh generation, mesh adaptation, numerical analysis, and graphical visualization is huge. The problem escalates and the computational power of modern computer systems increases rapidly, hence, recent software development requires the use of parallel algorithms with distributed data. All these applications undoubtedly have a common set of needs for representing and manipulating distributed unstructured meshes. The typical infrastructure needed by these applications is a set of data structures for representing mesh and software mechanisms for accessing and modifying mesh data. A mesh representation consists of topological mesh entities (vertices, edges, faces, cells) and topological adjacencies (interentity connections). The combination of a mesh representation and a set of access mechanisms for mesh data is called a mesh framework or meshing infrastructure. Although the need of common infrastructure to enable the rapid and efficient development of mesh based programs is obvious, no single framework is considered flexible enough and commonly accepted. One reason for scarcity of general meshing infrastructures is that the needs of the various meshing and analysis applications vary widely and there is little agreement on computational efficiency of different mesh data representation. Therefore, a large number of mesh representations are in use in the computational community each tailored to a specific application. Some simple numerical analysis programs use only a minimal representation consisting of elements (quads, tetrahedra, etc.) defined by nodes (or points). Other more sophisticated applications like complex finite volume schemes for general polyhedral meshes find it useful to exploit a much wider mesh representation consisting of a full set of mesh entities with an extensive set of topological adjacencies [1]. Therefore, to gain widespread acceptance it is important to have a full mesh framework which allows applications to access all types of mesh entities. At the same time, the infrastructure should be lightweight and efficient to have sufficient utilities for real world applications.

[1]Institute of Numerical Mathematics RAS, Moscow, Russia
[2]Stanford University, Stanford, USA

Fortunately, the need for general parallel meshing infrastructure is increasingly being recognized and several development efforts have been introduced in the last few years. These include the following packages: MSTK (Mesh Toolkit) [2], STK (Sierra Toolkit) [3], MOAB (A Mesh-Oriented datABase) [4, 5], FMDB (Flexible distributed Mesh DataBase) [6]. Some of these packages do not support dynamic modification of the mesh, some of them do not provide enough parallel functionality like several layers of ghost cells and some of them are still not reliable. Due to lack of appropriate parallel mesh framework for complex numerical analysis applications in early 2010s our group decided to design a convenient framework based on MSTK and MOAB which later evolved to the development of new parallel platform with flexibility and efficiency in mind – INMOST (Integrated Numerical Modeling Object-oriented Supercomputing Technologies).

INMOST is a newly developed, flexible, and efficient numerical analysis framework that provides application developers low-level infrastructure for reading, writing, creating, manipulating, and partitioning distributed unstructured meshes without having to design and implement their own mesh data structures. INMOST provides a convenient infrastructure for matrix assembling hence simplifying rapid development of discretization techniques. INMOST also provides framework for parallel linear system solvers including third party solver packages PETSc [7, 8] and Trilinos [9]. Newly developed parallel linear system solvers may be easily incorporated in or built upon INMOST infrastructure.

In this paper a brief description of INMOST is provided including details on design of the framework and software mechanisms for interacting with it. INMOST is in active development and some of the capabilities are not described here since their functionality may change in near future. However, the core functionality is considered stable and is described in this paper. INMOST is currently used to implement several applications such as safety analysis of nuclear and radioactive facilities [10], free surface computational fluid dynamics modeling [11], and black oil modeling [12] within Institute of Numerical Mathematics RAS Nuclear Safety Institute RAS and Stanford University. INMOST is currently available to general public under Modified BSD License at http://www.inmost.org/.

1. Platform structure

Only one general assumption on the mesh type is in place. The volume mesh is considered to be a conformal mesh with arbitrary polyhedral cells, i.e. any two cells either do not have common points or have one common vertex: either have one common edge or have one common face. INMOST platform have a modular structure allowing development of new modules and interfacing with third party software packages. The two core modules, mesh framework module and linear system solver module, will be covered in the current paper.

INMOST mesh is a dynamic distributed database representing unstructured grid. Operations of mesh modification and parallel mesh redistribution for load balancing require special data structure which is capable of data relocation, pruning, and compactification. Fast data access is one of the key features of INMOST. The direct memory access interface is used avoiding unnecessary data copying. This requires special attention to platform implementation ensuring continuous direct element data access during mesh modification at least until the mesh element is moved or deleted. Memory fragmentation during massive mesh modifications becomes a significant concern. INMOST platform utilizes a set of average size memory blocks for big data arrays. Each block is considered nonrelocatable, hence, ensuring continuous direct access the

blocks may be freed only during data compactification. INMOST implementation of mesh data structure is simple, flexible, and memory efficient with wide range of supported functionality. The detailed analysis of different mesh representations and supporting algorithms was performed by R. Garimella in his work [13]. INMOST mesh framework is based on full mesh representation with circular adjacencies vertex–edge–face–cell–vertex providing the balance between memory requirements and parallel algorithms efficiency (fig. 1).

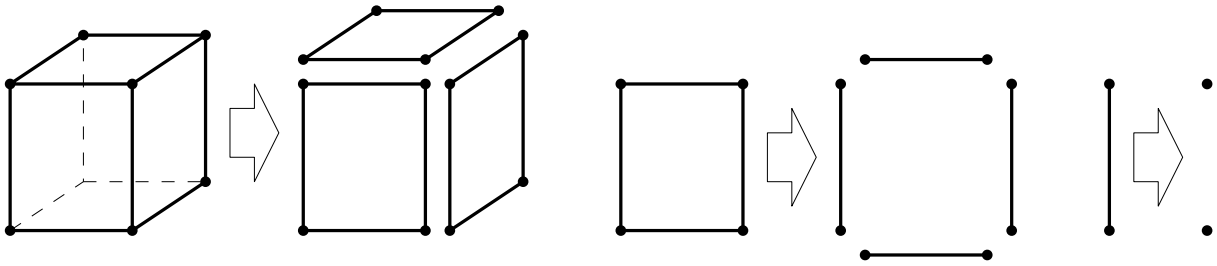


Figure 1. Downstream hierarchical representation of mesh elements

INMOST platform provides infrastructure for parallel grid generation. The user may rely on internal topology consistency checks and geometry calculators. The user decides which type of computed metric data for each type of mesh entities should be precomputed, cached, and updated on modification. The minimalistic parallel structured mesh generator will be presented below. Optionally the user can import and export the mesh in commonly used formats; internal cross architecture portable binary format is used to store the full set of mesh features.

One of the key points in parallel implementation of numerical analysis applications is the idea of domain decomposition and overlapping grids with ghost cells [14]. Ghost cells on one process are cells mirroring the corresponding cells from another process. Different discretization schemes may require different types and width of overlapping layers. In INMOST each mesh entity have exactly one process owner and a specific state for each sharing process: a “owned” one for entities which is only owned by the current process, a “shared” one for own entities which have copies on other processes, and a “ghost” one for copies of entities from other processes. Each shared entity also stores the list of processes it is copied to. INMOST automatically computes and distributes the ghost cells given the number of layers and connectivity type: neighbours through the faces, edges or vertices. Special attention is given to handling of multiple ghost layers, since cross-process transfers occur commonly. The user can also provide an explicit ghost map for specific applications.

The second key point in effective parallel computation is the load balancing. INMOST provides flexible interface for repartitioning and redistribution of the mesh. The user chooses one of the internal partitioners, the external widely used partitioners Zoltan [15] and Parmetis [16], or provides the partitioning map explicitly. Redistribution process effectively utilizes the ghost layers if any of them have been computed in advance and ensures that the ghost layers structure is preserved after redistribution.

INMOST platform provides flexible infrastructure for organization of mesh entities into hierarchical sets and associate user data with mesh entities. The user data is identified by named tags. Each tag may be associated with vertices, edges, faces, cells, sets, entire mesh or any combinations of them (fig. 2). Each tag stores real, integer or byte data or references to elements. Fixed and dynamic length arrays are natively supported. Tags may be dense, i.e. all appropriate mesh entities have the associated tag, or sparse, i.e. only some of the entities store the tag data. Markers are used in the same way as a boolean type dense tags. The tags data is

mirrored on demand to ghost cells, in addition INMOST supports common reduce operations to gather data from ghost cells back to owner.

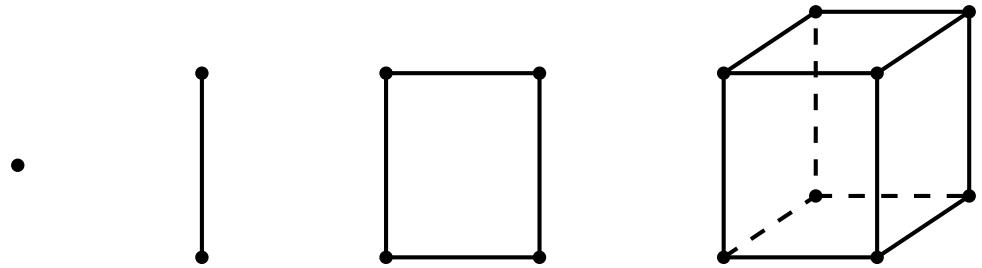


Figure 2. Mesh entities: vertex, edge, face and cell

INMOST solver class provides methods for sparse matrix assembling without prior knowledge of matrix sparsity structure. The linear solver infrastructure may be used to develop and implement specific iterative linear solver with special preconditioners. Several internal solvers are provided natively and convenient interfaces to PETSc and Trilinos solver packages are included as well.

The detailed algorithms and data structures are presented in [17].

2. User interface

```
#include "inmost.h"
using namespace INMOST;
int main(int argc, char *argv[])
{
  // Initialize INMOST activities
  Solver::Initialize(&argc,&argv,"");
  Mesh::Initialize(&argc,&argv);
  Partitioner::Initialize(&argc,&argv);
  ...
  // Finalize INMOST activities
  Partitioner::Finalize();
  Solver::Finalize();
  Mesh::Finalize();
  return 0;
}
```

Figure 3. Initialization and finalization of INMOST modules

The INMOST code is written in C++ language, it is a cross-platform and may be built using wide range of compilers. Modular structure of INMOST code allows the user to enable and disable optional components including interfaces to third party software. The code is organized in a way that the user does not need to exploit any of MPI specific procedures in order to create a parallel application. If any INMOST module is used it should be properly initialized and finalized before and after of its use respectively; see Fig. 3 for examples of several modules usage.

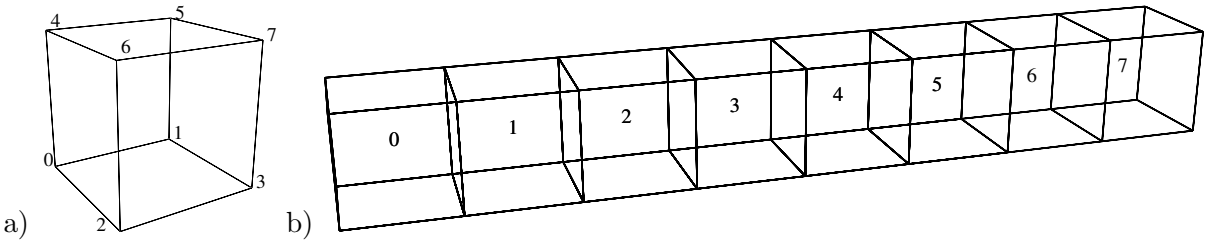


Figure 4. Minimalistic parallel cubic grid: a) one cell with local vertices, b) distributed grid with NP=8

```
Mesh *m = new Mesh();
int rank = m->GetProcessorRank(); // Get process rank
int size = m->GetProcessorsNumber(); // Get number of processes
ElementArray<Node> verts(m); // Create local vertices
for(int k=0; k<2; k++)
  for(int j=0; j<2; j++)
    for(int i=0; i<2; i++)
    {
      double xyz[3]={rank+i, j, k}; // Define node coordinates
      verts.push_back(m->CreateNode(xyz)); // Add new vertex
    }
// Define face connectivity template for cubic cell, rf. Fig. 4a
const int face_nodes[24] = {0,4,6,2, 1,3,7,5, 0,1,5,4,
                            2,6,7,3, 0,2,3,1, 4,5,7,6};
const int num_nodes[6]   = {4, 4, 4, 4, 4, 4};
m->CreateCell(verts,face_nodes,num_nodes,6); // Create the cubic cell
m->ResolveShared(); // Resolve duplicate nodes
m->Save("mesh.pvtk"); // Export mesh to parallel VTK format
delete m;
```

Figure 5. Minimalistic parallel mesh generation

In order to create a parallel mesh the user should set an MPI communicator, an alias INMOST_MPI_COMM_WORLD is provided for convenience by default. Once the communicator is set the process may obtain its rank and the total number of processes becomes known. We demonstrate minimalistic distributed mesh generation process below (see Figs. 4b, 5). One way to create a polyhedral cell is to define all nodes, all edges using nodes, all faces using edges, and a cell using faces. Alternatively short way is to define nodes and create a cell using connectivity template for its faces. In our example we use the second way: we define 8 nodes and create a cell with six faces, each face containing four nodes and their indices are prescribed by face_nodes template (see Fig. 4a). Once local grids are created we use ResolveShared() procedure to resolve duplicate nodes and assign global IDs to mesh entities. The distributed mesh is exported using Save() method.

The same idea is used to construct the distributed rectangular mesh, the only difference is that more vertices are defined and the cells are constructed in loop. The INMOST source package provides “GridGen” example which creates a distributed rectangular mesh for a unit cube.

This example can also be used to create a prismatic mesh. The readers are refered to the example source code for further details.

```
Mesh *m = ... // create or load mesh from file
m->ExchangeGhost(1,FACE);
Mesh::GeomParam table;
table[BARYCENTER] = CELL;
table[NORMAL] = FACE;
table[MEASURE] = CELL | FACE;
m->PrepareGeometricData(table);
Solver::Matrix A;
Solver::Vector b;
// Iterate over all faces
for(Mesh::iteratorFace f = m->BeginFace(); f != m->EndFace(); ++f)
{
  Cell r1 = f->BackCell(), r2 = f->FrontCell();
  // Skip face if cells are ghosts or not available
  if( (!r1->isValid() || r1->GetStatus() == Element::Ghost) &&
      (!r2->isValid() || r2->GetStatus() == Element::Ghost) ) continue;
  if (r1->isValid() && r2->isValid()) { // Internal face case
    double f_area = f->Area(); // Get the face area
    double f_nrm[3], r1_cnt[3], r2_cnt[2];
    f->Normal(f_nrm); // Get the face normal
    r1->Barycenter(r1_cnt); // Get the barycenter of the cell r1
    r2->Barycenter(r2_cnt); // Get the barycenter of the cell r2
    double coef = ... // Compute flux coefficients
                      // using f_area, f_nrm, r1_cnt, r2_cnt
    int id1 = r1->GlobalID(), id2 = r2->GlobalID(); // Get global IDs
    // Fill matrix coefficients only for normal cells
    if( r1->GetStatus() != Element::Ghost )
      A[id1][id1] += -coef, A[id1][id2] += coef;
    if( r2->GetStatus() != Element::Ghost )
      A[id2][id1] += coef, A[id2][id2] += -coef;
  } else {
    ... // Boundary face case
  }
}
// Iterate over all cells
for( Mesh::iteratorCell c = m->BeginCell(); c != m->EndCell(); ++c )
  if( c->GetStatus() != Element::Ghost )
    b[c->GlobalID()] += c->Volume()*c->Mean(rhs, 0); // Integrate rhs()
```

Figure 6. Assemble matrix of linear system

Different discretization techniques result in matrix assembling. We demonstrate the template for finite volume (FV) scheme shown in Fig. 6. The detailed example is packed within

```
Solver S(Solver::INNER_ILU2); // Specify the linear solver
S.SetMatrix(A); // Compute the preconditioner for the original matrix
S.Solve(b,x);   // Solve the linear system with the preconditioner
Tag phi = m->CreateTag("Solution", DATA_REAL, CELL, NONE, 1);
for(Mesh::iteratorCell c = m->BeginCell(); c != m->EndCell(); ++c)
  if( c->GetStatus() != Element::Ghost )
    c->Real(phi) = x[c->GlobalID()];
```

Figure 7. Solve linear system and attach solution to mesh cells

INMOST source package as "FVDiscr" example. In general case matrix elements depend on the neighboring ones, thus, a layer of ghost cells is usually needed. ExchangeGhost() method is used to create one or several layers of ghost cells. We use two point flux approximation in FV scheme, hence, for each internal face we need its area and barycenters of neighboring cells. We also need cell volumes for right-hand side calculation. INMOST can precompute and cache the needed geometric information using PrepareGeometricData() method. Several entity iterators are available in INMOST. In our FV scheme we iterate over all faces and compute matrix coefficients for neighboring cells BackCell() and FrontCell(). Global cell identificators GlobalID() are used as indices to create matrix and right-hand side vector.

Once the matrix is assembled one can utilize internal BiCGStab(L) solver with second order ILU factorization as preconditioner. The code example is presented in Fig. 7. The computed solution is stored in tag "Solution", a single real value is attached to all cells. A more detailed example of linear system solution is presented in "MatSolve" example from INMOST source code.

More complicated examples and test cases are bundled in INMOST package and demonstrate in more detail mesh generation, mesh partitioning and redistribution, matrix assembling for FV scheme of diffusion problem and linear system solution using different solvers and packages. The user is advised to consult with online documentation available on project site and take a look at unit tests for mesh and solver modules.

3. Numerical experiments

In this section several performance tests are presented using the code introduced above and included as code examples in INMOST package.

The code is used to solve the problem $-\nabla \cdot (K \nabla U) = f$ with Dirichlet boundary conditions, where K is unit tensor and the right-hand side f is computed from the exact solution: $U = \sin(\pi x)\sin(\pi y)\sin(\pi z)$. The parallel code creates a rectangular grid in unit cube, this stage is called "GridGen" below. The simplest two-point FVM scheme is used to assemble local matrices. Using ghost cells effectively links local matrices in a global matrix. This matrix assembly stage is refered to as "Assemble" below. At the final stage the linear system is solved, namely "MatSolve" stage below.

To perform parallel numerical experiments, two parallel computer systems were used: the INM RAS computer cluster and the "Lomonosov" computer cluster. We exploited the nodes of the "x6core" queue of the INM RAS cluster with the total of 12 nodes. These are Xeon X5650@2.67GHz nodes with 24 GB of memory and 12 cores per node. We also exploited the nodes of "regular4" queue of Lomonosov cluster with the total of 64 nodes. These are Xeon

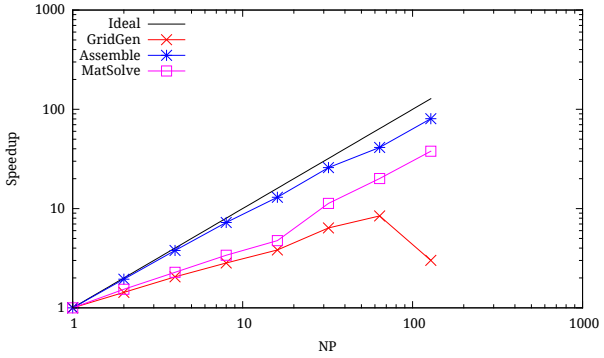


Figure 8. Strong scalability test on INM RAS computer cluster, speedup graph

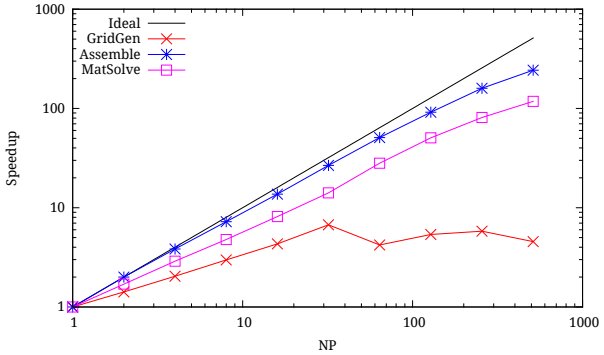


Figure 9. Strong scalability test on "Lomonosov" computer cluster, speedup graph

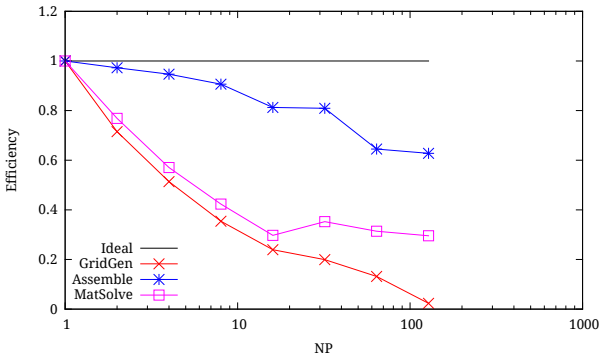


Figure 10. Strong scalability test on INM RAS computer cluster, efficiency graph

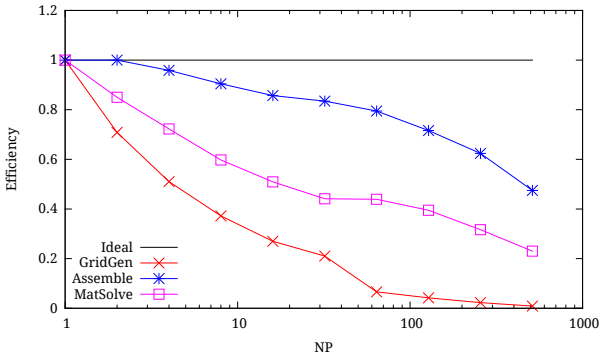


Figure 11. Strong scalability test on "Lomonosov" computer cluster, efficiency graph

X5570@2.93Ghz nodes with 12 GB of memory and 8 cores per node. In both cases Intel Compilers with Intel MPI were used.

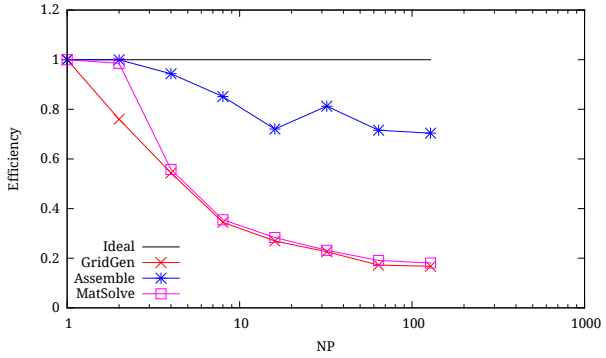


Figure 12. Weak scalability test on INM RAS computer cluster, efficiency graph

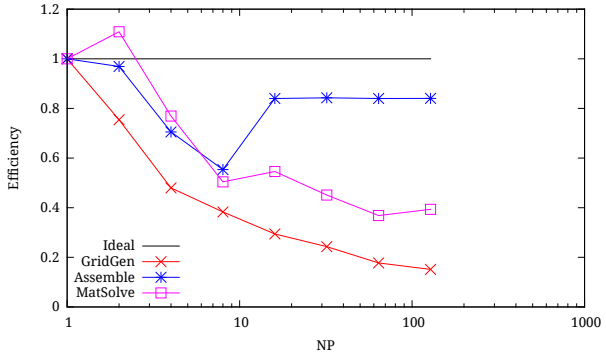


Figure 13. Weak scalability test on "Lomonosov" computer cluster, efficiency graph

Table 1. Strong scalability test on INM RAS computer cluster

	GridGen			Assemble			MatSolve		
NP	T	S	E	T	S	E	T	S	E
1	11.435	1.00	1.000	19.029	1.00	1.000	28.841	1.00	1.000
2	7.994	1.43	0.715	9.785	1.94	0.972	18.759	1.54	0.769
4	5.566	2.05	0.514	5.025	3.79	0.947	12.641	2.28	0.570
8	4.035	2.83	0.354	2.624	7.25	0.906	8.519	3.39	0.423
16	2.985	3.83	0.239	1.464	13.00	0.813	6.060	4.76	0.297
32	1.793	6.38	0.199	0.735	25.89	0.809	2.557	11.28	0.352
64	1.356	8.43	0.132	0.461	41.29	0.645	1.436	20.09	0.314
128	3.803	3.01	0.023	0.237	80.37	0.628	0.762	37.84	0.296

Both strong and weak scalability tests were performed on both clusters. The strong scalability test was performed using uniform $96 \times 96 \times 96$ cubic grid on INM RAS cluster (fig. 8, fig. 10), and uniform $100 \times 100 \times 100$ cubic grid on "Lomonosov" cluster (fig. 9, fig. 11). Total times, speedups, and efficiencies are presented in tab. 1 and tab. 2. Weak scalability test was performed using $64 \times 64 \times 64$ cubic grid on each process on INM RAS cluster (fig. 12), and $50 \times 50 \times 50$ cubic grid on each process on "Lomonosov" cluster (fig. 13). Total times and efficiencies are presented in tab. 3 and tab. 4.

Numerical experiments have showen reasonable scalability of "Assemble" and "MatSolve" examples during strong scalability tests. The "Assemble" example also results in high efficiency during weak scalability tests. Minor reduction in efficiency observed with $NP = 8$ on

Table 2. Strong scalability test on "Lomonosov" computer cluster

	GridGen			Assemble			MatSolve		
NP	T	S	E	T	S	E	T	S	E
1	15.882	1.00	1.000	28.552	1.00	1.000	64.849	1.00	1.000
2	11.204	1.42	0.709	14.279	2.00	1.000	38.129	1.70	0.850
4	7.783	2.04	0.510	7.446	3.83	0.959	22.438	2.89	0.723
8	5.335	2.98	0.372	3.946	7.24	0.904	13.561	4.78	0.598
16	3.677	4.32	0.270	2.082	13.71	0.857	7.960	8.15	0.509
32	2.359	6.73	0.210	1.069	26.71	0.835	4.594	14.12	0.441
64	3.768	4.22	0.066	0.561	50.85	0.795	2.308	28.10	0.439
128	2.956	5.37	0.042	0.311	91.68	0.716	1.283	50.55	0.395
256	2.737	5.80	0.023	0.179	159.76	0.624	0.800	81.03	0.317
512	3.488	4.55	0.009	0.117	243.08	0.475	0.550	117.94	0.230

Table 3. Weak scalability test on INM RAS computer cluster

	GridGen		Assemble		MatSolve	
NP	T	E	T	E	T	E
1	3.484	1.000	5.589	1.000	7.469	1.000
2	4.583	0.760	5.592	0.999	7.580	0.985
4	6.406	0.544	5.925	0.943	13.393	0.558
8	10.106	0.345	6.566	0.851	21.050	0.355
16	12.924	0.270	7.757	0.720	26.350	0.283
32	15.409	0.226	6.876	0.813	32.199	0.232
64	20.233	0.172	7.809	0.716	39.109	0.191
128	20.767	0.168	7.938	0.704	41.319	0.181

Table 4. Weak scalability test on "Lomonosov" computer cluster

	GridGen		Assemble		MatSolve	
NP	T	E	T	E	T	E
1	2.042	1.000	3.364	1.000	7.037	1.000
2	2.706	0.754	3.471	0.969	6.346	1.109
4	4.254	0.480	4.770	0.705	9.149	0.769
8	5.329	0.383	6.078	0.554	13.953	0.504
16	6.935	0.294	4.005	0.840	12.892	0.546
32	8.396	0.243	3.993	0.843	15.606	0.451
64	11.507	0.177	4.004	0.840	19.111	0.368
128	13.485	0.151	4.004	0.840	17.883	0.394

"Lomonosov" cluster (fig. 13) can be attributed to the limited memory access bandwidth on one node. Starting with $NP = 16$ the number of physical nodes increases and the overall

bandwidth increases as well. Poor performance of "GridGen" example can be attributed to mesh generator design flaw. The final step of ResolveShared() procedure involves multiple communications for huge grids. A better approach would be to generate coarse grid first, resolve shared entities, and only then refine the mesh on each processor.

Conclusions

INMOST program platform is presented which allows a user to work with distributed data on general meshes. Internal platform structure is addressed to showing flexibility and efficiency of the infrastructure. Several user interface examples are used for minimalistic demonstration purposes of mesh generation, matrix assembling and linear system solution. Numerical results demonstrate reasonable scalability of matrix assembly and linear system solution stages.

The work is partially supported by the Russian Foundation for Basic Research (RFBR) grants No.14-01-00830 and No.15-35-20991.

References

1. Danilov AA, Vassilevski YV. A monotone nonlinear finite volume method for diffusion equations on conformal polyhedral meshes. Russian Journal of Numerical Analysis and Mathematical Modelling. 2009;24(3):207–227.

2. Garimella RV. MSTK – A Flexible Infrastructure Library for Developing Mesh Based Applications. In: 13th International Meshing Roundtable, September 19-22, 2004, Williamsburg, Virginia, USA, Proceedings; 2004. p. 203–212.

3. Edwards HC, Williams AB, Sjaardema GD, Baur DG, Cochran WK. SIERRA Toolkit Computational Mesh Conceptual Model. Technical Report SAND2010-1192, Sandia National Laboratories; 2010.

4. Tautges TJ. MOAB-SD: Integrated Structured and Unstructured Mesh Representation. Engineering With Computers. 2004;20(3):286–293.

5. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C. MOAB: A Mesh-Oriented Database. Technical Report SAND2004-1592, Sandia National Laboratories; 2004.

6. Seol ES. FMDB: flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis, Ph.D. Thesis, Rensselaer Polytechnic Institute; 2005.

7. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, et al. PETSc users manual. Argonne National Laboratory, ANL-95/11 - Revision 3.5; 2014.

8. Balay S, Gropp WD, McInnes LC, Smith BF. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In: Modern Software Tools in Scientific Computing, Birkhäuser Press; 1997. p. 163–202.

9. Heroux MA, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, et al. An overview of the Trilinos project. ACM Transactions on Mathematical Software. 2005;31(3):397–423.

10. Kapyrin I, Konshin I, Kopytov G, Nikitin K, Vassilevski Y. Hydrogeological modeling in radioactive waste disposal safety assessment using the GeRa code. Russian Supercomputing Days: Proceedings of the international conference (September 28-29, 2015, Moscow, Russia). Moscow State University; 2015. p. 122–132. Russian.

11. Terekhov KM, Nikitin KD, Olshanskii MA, Vassilevski YV. A semi-Largangian method on dynamically adapted octree meshes, to appear in Rus.J.Num.Anal.Math.Model.

12. Konshin I, Kaporin I, Nikitin K, Vassilevski Y. Parallel linear systems solution for multiphase flow problems in the INMOST framework. Russian Supercomputing Days: Proceedings of the international conference (September 28-29, 2015, Moscow, Russia). Moscow State University; 2015. p. 96–103.

13. Garimella RV. Mesh Data Structure Selection for Mesh Generation and FEA Applications. International Journal of Numerical Methods in Engineering. 2002;55(4):451–478.

14. Bertsekas DP, Tsitsiklis JN. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall; 1989.

15. Boman EG, Catalyurek UV, Chevalier C, Devine KD. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: partitioning, ordering, and coloring. Scientific Programming. 2012;20(2):129–150.

16. Schloegel K, Karypis G, Kumar V. Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience. 2002;14(3):219–240.

17. Vassilevski Y, Konshin I, Kopytov G, Terekhov K. INMOST – a software platform and a graphical environment for development of parallel numerical models on general meshes. Moscow State Univ. Publ., Moscow; 2013. Russian.

DOI: 10.14529/jsfi150405

Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems

M. Abalenkovs[1], *A. Abdelfattah*[2], *J. Dongarra* [1,2,3], *M. Gates*[2], *A. Haidar*[2], *J. Kurzak*[2], *P. Luszczek*[2], *S. Tomov*[2], *I. Yamazaki*[2], *A. YarKhan*[2]

We present a review of the current best practices in parallel programming models for dense linear algebra (DLA) on heterogeneous architectures. We consider multicore CPUs, stand alone manycore coprocessors, GPUs, and combinations of these. Of interest is the evolution of the programming models for DLA libraries – in particular, the evolution from the popular LAPACK and ScaLAPACK libraries to their modernized counterparts PLASMA (for multicore CPUs) and MAGMA (for heterogeneous architectures), as well as other programming models and libraries. Besides providing insights into the programming techniques of the libraries considered, we outline our view of the current strengths and weaknesses of their programming models – especially in regards to hardware trends and ease of programming high-performance numerical software that current applications need – in order to motivate work and future directions for the next generation of parallel programming models for high-performance linear algebra libraries on heterogeneous systems.

Keywords: Programming models, runtime, HPC, GPU, multicore, dense linear algebra.

Introduction

Parallelism in today's computer architectures is pervasive – not only in systems from large supercomputers to laptops, but also in small portable devices like smart phones and watches. Along with parallelism, the level of heterogeneity in modern computing systems is also gradually increasing. Multicore CPUs are often combined with discrete high-performance accelerators like graphics processing units (GPUs) and coprocessors like Intel's Xeon Phi, but are also included as integrated parts with system-on-chip (SoC) platforms, as in the AMD Fusion family of application processing units (APUs) or the NVIDIA Tegra mobile family of devices. To extract full performance from systems like these, the heterogeneity makes the parallel programming for technical computing problems extremely challenging.

Furthermore, when considering the future of research and development on parallel programming models, the two factors that should be looked at first are the projected changes in system architectures that will define the landscape on which HPC software will have to execute, and the nature of the applications and application communities that this software will ultimately have to serve. In the case of system architectures, we can get a reasonable picture of the road that lies ahead by examining the roadmaps of major hardware vendors, as well as the next generation of supercomputing platforms that are in the works, discussed in Section 1. In the case of applications, we can extract trends of what may be needed from applications of high current interest like big-data analytics, machine learning, and more, in Section 5.

A traditional way to take advantage of parallelism without tedious parallel programming is through the use of already parallelized HPC libraries. Among them, linear algebra (LA) libraries, and in particular DLA, are of high importance since they are fundamental to a wide range of scientific and engineering applications. Hence, these applications will not perform well unless LA

[1]University of Manchester, Manchester, UK
[2]University of Tennessee, Knoxville, TN, USA
[3]Oak Ridge National Laboratory, Oak Ridge, TN, USA

libraries perform well. Thus, by concentrating on best practices in parallel programming models for DLA on heterogeneous architectures, we will indirectly address other areas, as long as the DLA libraries discussed and their programming models are interoperable with third party tools and standards.

To describe how to program parallel DLA, we first focus on the generic parallel programming models today (Section 2), and second, on the evolution in the programming models for DLA libraries – from the popular LAPACK and ScaLAPACK to their modernized counterparts in PLASMA, for multicore CPUs, and MAGMA, for heterogeneous architectures, as well as in other programming models and libraries (Section 3). The current best practices are summarized by the task based programming model, reviewed in Section 4. Besides providing insights into the programming techniques of the libraries considered, we outline our view of the current strengths and weaknesses of their programming models – especially in regards to hardware trends and ease of programming high-performance numerical software that current applications need – in order to motivate work and future directions for the next generation of parallel programming models for high-performance LA libraries on heterogeneous systems (Section 5).

1. Hardware trends in heterogeneous system designs

Some future trends in the heterogeneous system designs are evident from USA's DOE plans for the next generation of supercomputers. The aim is to deploy three different platforms by 2018, each with over 150 petaflops of peak performance [18]. Two of them, named Summit and Sierra, will be based on IBM OpenPOWER and NVIDIA GPU accelerators, and the third, Aurora, will be based on the Xeon Phi platform. Summit and Sierra will follow the hybrid computing model by coupling powerful latency-optimized processors with highly parallel throughput-optimized accelerators. They will rely on IBM Power9 CPUs, NVIDIA Volta GPUs, an NVIDIA NVLink interconnect to connect the hybrid devices within each node, and a Mellanox Dual-Rail EDR Infiniband interconnect to connect the nodes. The Aurora system, on the other hand, will offer a more homogeneous model by utilizing the Knights Hill Xeon Phi architecture, which, unlike the current Knights Corner, will be a stand-alone processor and not a slot-in coprocessor, and will also include integrated Omni-Path communication fabric. All platforms will benefit from recent advances in 3D-stacked memory technology, and promise major performance improvements:

- CPU memory bandwidth is expected to be between 200 GB/s and 300 GB/s using HMC;
- GPU memory bandwidth is expected to approach 1 TB/s using HBM;
- GPU memory capacity is expected to reach 60 GB (NVIDIA Volta);
- NVLink is expected to deliver from 80 up to 200 GB/s of CPU-to-GPU bandwidth;
- In terms of computing power, the Knights Hill is expected to be between 3.6 and 9 teraflops, while the NVIDIA Volta is expected to be around 10 teraflops.

The hybrid computing model is here to stay, and memory systems will become ever more complicated. Moreover, the interconnection technology will be seriously lagging behind the computing power. Aurora's nodes may have 5 teraflops of computing power with a network injection bandwidth of 32 GB/s; Summit's and Sierra's nodes are expected to have 40 teraflops of computing power and a network injection bandwidth of 23 GB/s. This creates a gap of two orders of magnitude for Aurora and three orders of magnitude for Summit and Sierra, leaving data-heavy workloads in the lurch and motivating the search for algorithms that minimize data movement, and programming models that facilitate their development.

The gap between compute power and interconnect levels will continue to widen, diversify, and deepen not just between nodes, as pointed out, but also within the node's entire memory hierarchy, as evident from hardware developments such as a significant increase in the *last level cache* (LLC) size in the CPU, and L2 cache in the GPU, use of NVLink with a higher interconnect bandwidth, and 3D-stacked memories (see also Figure 1, Left). These trends will thus require some support from the programming model for hierarchical tasks (and their communications) for blocking computations over the memory hierarchies [44] in order to reduce data movement. In fact, the community agrees that the biggest challenges to future application performance lie with efficient node-level execution that can use all the resources in the node [58], thus motivating this paper's focus to be on the programming models at the node level.

2. General purpose parallel programming models

A thorough survey of general purpose parallel programming models is beyond the scope of this writing and would exceed space constraints imposed on the content. However, there already exist overview-style publications that provide a comprehensive list of programming models for manycore [58] and heterogeneous hardware [42]. The success of the accelerator hardware resulted in an unfortunate proliferation of software solutions for easing the programming burden that the new systems bring to bear. Even with the recent coalescing of the plethora of products around major standardization themes, the end user is still left with the choice of one of many *open* specifications: OpenCL [38], OpenMP 4 [48, 49], and OpenACC 2 [46]. Despite similarity in the name, there is a vast disparity between the levels of abstraction that these standards offer to the programmer. OpenCL defines a software library that exposes a very low-level view of the hardware. The other two standards, OpenMP and OpenACC, are language-based solutions that hide tremendous amounts of detail and offer a much more succinct means of expressing the offloading paradigm of programming. An additional consideration is the fact that many algorithms, and in particular, algorithms in the area of DLA, are designed to use the Basic Linear Algebra Subprograms (BLAS) [20] standard, and therefore it is important to note that, as of today, there is no fully compliant BLAS implementation in OpenACC or OpenMP. Instead, there is a possibility of calling vendor BLAS which might be, but not necessarily are, implemented with OpenCL. Thus, from the very specific perspective of library development, only OpenCL offers a sufficient breadth of features essential for performance, portability, and maintainability, without the burden of extraneous software dependencies. Unfortunately, the actual performance achieved by OpenCL implementations on some platforms does not match less portable interfaces that are perfected for the hardware by the vendor [21]. A lower level, and consequently closer to hardware, model is based on dataflow and streams [51]. On the NVIDIA hardware, the CUDA Toolkit a is widely known and used software stack that exposes the underlying hardware through a simple abstraction of streams that run on processing pipelines that are efficiently scheduled with the help of a hardware-based scheduler. On the Xeon Phi hardware accelerator from Intel, the streaming model for offload computing is also available [45] with the additional advantage of offloading across not just the Intel discrete accelerators but also the CPU processors. Finally for completeness, we would also like to mention C++ AMP [13, 23], which is a standard that targets Microsoft's DirectX and DirectCompute software stacks. AMP is much closer conceptually to OpenMP and OpenACC in terms of offering a single source code solution with restricted syntax for the offload regions to account for limited capabilities of the accelerator hardware. Note that OpenMP is much more permissive in terms of syntax due to Xeon Phi's much more CPU-like

design. The recent porting efforts might make AMP much more relevant outside of the Windows and Visual Studio ecosystems as there is a port to LLVM that is advancing in functionality due to efforts by MultiCoreWare and overseen by the HSA Foundation[4].

3. Special purpose parallel programming models

3.1. LAPACK and ScaLAPACK

The LAPACK's programming model [4] is based on expressing algorithms in terms of BLAS calls, described below. Subsequently, LAPACK can achieve high efficiency, provided that highly efficient machine-specific BLAS implementations are provided, e.g., by the manufacturer. Since the 1980s this model has turned out to be very successful for cache-based shared-memory vector and parallel processors with multi-layered memory hierarchies.

ScaLAPACK is the distributed-memory implementation of LAPACK, where BLAS is replaced by a parallel BLAS (PBLAS). This design makes it possible for the ScaLAPACK routines to be quite similar to their LAPACK counterparts. Efficiency within a node is extracted through the use of BLAS. Load balance and scalability is largely achieved by the way matrices are distributed over the processes – namely, in a 2D block cyclic distribution [11].

3.1.1. BLAS

The original Level 1 BLAS [40] and an extended, Level 2 BLAS for matrix-vector operations [19] were adopted as standard and used in a wide range of numerical software, including LINPACK [16]. Unfortunately, while successful for the vector-processing machines at the time, Level 2 BLAS was not a good fit for the cache-based machines that emerged in the 1980s because the operations are memory bound and thus, do not use the multi-layered memory hierarchies; thereby spending too much time moving data instead of doing useful floating-point operations. To effectively use caches, it was preferable to express computations as matrix-matrix operations. Matrices were split into small blocks so that basic operations were performed on blocks that fit into cache memory. This approach avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of operations to data movement. Subsequently, Level 3 BLAS was proposed [20], covering the main types of matrix-matrix operations, and LINPACK was redesigned into LAPACK to use the new Level 3 BLAS where possible.

To account for the deep memory hierarchies today, efficient Level 3 BLAS implementations feature multilevel blocking where the matrix-matrix computations are split hierarchically into blocks that fit into corresponding levels of the memory hierarchy. Thus, we point out that a programming model based on using Level 3 BLAS provides support for hierarchical tasks, as needed and highlighted in Section 1, and is still the best parallel programming model (or at least the main component in custom models; discussed below) for DLA at the present time.

3.1.2. PBLAS

Custom parallel programming models for DLA are usually built on top of BLAS, and their purpose is to ease parallel programming efforts and facilitate obtaining scalable high-performance. The use of BLAS is one main component in obtaining high-performance. Another, which is orthogonal to the benefits of using BLAS, is the support of hierarchical tasks for the

[4] http://www.hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/

memory hierarchies not covered by BLAS. In the case of PBLAS [12], this is the node-to-node interconnection layer, denoted as the *Remote CPU main memory* layer in Figure 1. PBLAS is still a very successful parallel programming abstraction for DLA on distributed-memory systems. As pointed out, the biggest challenges to future application performance lie within efficient node-level execution, where nodes are becoming highly parallel and heterogeneous, which is our main focus. With that in mind, we leave the distributed-memory case with PBLAS as an excellent example of how to organize DLA to capture one more level of hierarchy (internodal) over BLAS, and thus allowing coding algorithms for distributed-memory systems to look like the ones for shared-memory systems built on top of BLAS (as in LAPACK).

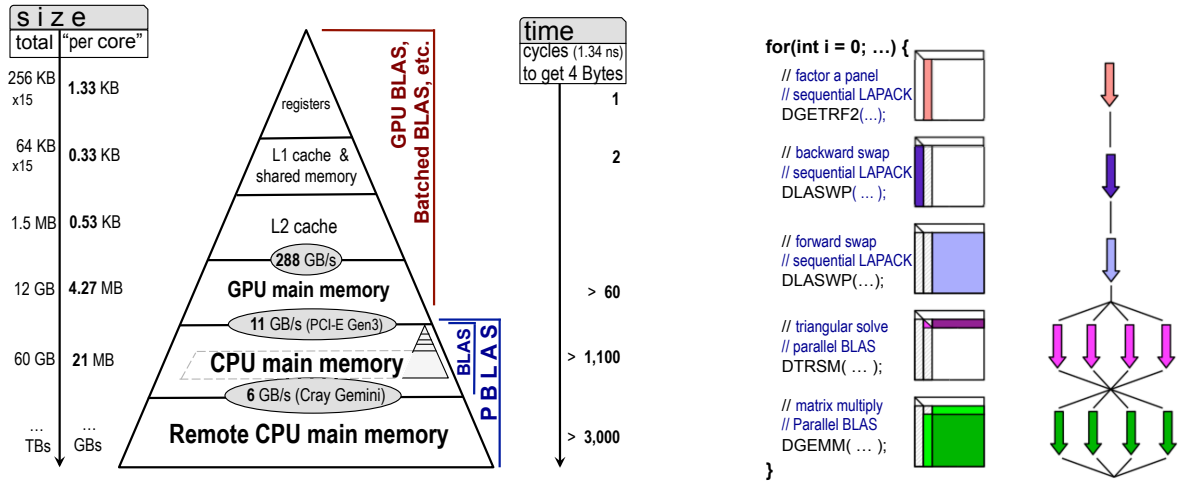


Figure 1. Left: Memory hierarchy of a CUDA core of an NVIDIA K40c GPU with 2,880 CUDA cores. **Right**: A fork-join parallelism example in an LU factorization using parallel BLAS.

3.2. The fork-join parallel model

The use of parallel BLAS – for either shared-memory systems (e.g., single GPU, or multicore CPUs system) or distributed-memory systems (as in PBLAS) – results in a parallel execution model often referred to as *fork-join*. The naming arises from the fact that in this model a sequence of BLAS calls is implicitly synchronized after each individual BLAS call (join), and the routines by themselves run in parallel (fork), as illustrated in Figure 1, Right. This is a powerful model as it is simple to use and develops scalable high-performance parallel algorithms. However, the overhead of synchronization and idle processors/cores can be large for small problems. This motivates the search for improved models such as the task approach in PLASMA, or the batched approach in MAGMA [28] for very small problems and even tiny problems (sub-vector/warp in size) that may require a few tasks to be grouped for execution on a single core [1].

3.3. PLASMA and the task approach

Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [2] was developed to address the performance deficiency of the LAPACK library on multicore processors. LAPACK's shortcomings stem from the fact that its algorithms are coded in a serial fashion, with parallelism only possible inside the set of BLAS, which forces a fork-join style of multithreading. In contrast, algorithms in PLASMA are coded in a way that allows for much more powerful dataflow parallelism [10, 39]. Currently, PLASMA supports a substantial subset of LAPACK's functionality,

including solvers for linear systems of equations, linear least squares problems, singular value problems, symmetric eigenvalue problems and generalized symmetric eigenvalue problems. It also provides a full set of Level 3 BLAS routines for matrices stored in tile layout and a highly optimized (cache-efficient and multithreaded) set of routines for layout translation [24]. PLASMA is based on three principles: tile algorithms, tile matrix layout and task-based scheduling, all of which are geared towards efficient multithreaded execution.

Initially, PLASMA routines were implemented using the *Single Program Multiple Data* (SPMD) programming style, where each thread executed a statically scheduled preassigned set of tasks, while coordinating with other threads using progress tables, with busy-waiting as the main synchronization mechanism. While seemingly very rudimentary, this approach actually produced very efficient, systolic-like, pipelined processing patterns, with good data locality and load balance. At the same time, the codes were hard to develop, error prone and difficult to maintain, which motivated a move towards dynamic scheduling.

The multicore revolution of the late 2000's brought the idea back and put it in the mainstream. One of the first task-based multithreading systems that received attention in the multicore era, was the Cilk programming language, originally developed at MIT in the 1990's. Cilk offers nested parallelism based on a tree of threads that is geared towards recursive algorithms. About the same time, the idea of superscalar scheduling gained traction, based on scheduling tasks by resolving data hazards in real time, in a similar way that superscalar processors dynamically schedule instructions on CPUs.

This superscalar technique was pioneered by a software project from the Barcelona Supercomputing Center, which went through multiple names as its hardware target was changing: GridSs, CellSs, SMPSs, OMPSs, StarSs, where "Ss" stands for superscalar [9, 22, 50]. A similar development was the StarPU project from INRIA, which applied the same methodology to systems with GPU accelerators, named for its capability to schedule work to "C"PUs and "G"PUs, hence the name *PU, transliterated into StarPU [7]. Yet another scheduler was developed at Uppsala University, called SuperGlue [53]. Finally, a system called QUARK was developed at the University of Tennessee [59], and used for implementing the PLASMA numerical library [2]. At some point, all the projects mentioned received extensions for scheduling in distributed memory.

The OpenMP community has been swiftly moving forward with standardization of new scheduling techniques for multicores. First, the OpenMP 3.0 standard [47] adopted the Cilk scheduling model, then the OpenMP 4.0 standard [48] adopted the superscalar scheduling model. Not without significance is the fact that the GNU compiler suite was also quick to follow with high quality implementations of the new extensions. Motivated by these recent developments, PLASMA is currently in the process of being completely ported to the OpenMP standard.

3.4. MAGMA and the hybrid approach

The MAGMA library implements hybrid DLA routines for heterogeneous systems using both CPUs and accelerators, such as a GPU or Intel Xeon Phi. To best utilize a heterogeneous system requires understanding the strengths and weaknesses of each processor. We will briefly review the architectures and how those impact computations.

GPUs use a *single instruction multiple thread* (SIMT) model. The computation is arranged in a grid of thread blocks, which is further divided into a grid of threads. Within each thread block, threads are executed in sets; for instance a set of 32 threads known as a *warp* in NVIDIA's CUDA architecture. In each thread block, threads are not independent, but in SIMT fashion

follow the same execution path. Threads may take different branches, but this incurs a significant penalty as all threads must wait for both sides of the branch to finish. Ideally, most of the time all threads execute the same instruction on different data.

As with CPUs, reusing data in registers and caches is important for maximizing the ratio of floating point operations to data transfers. This architecture means GPUs are most efficient at doing bulk computations with regular data access – exactly the kind of computations in dense matrix-matrix operations (Level 3 BLAS). For instance, the cuBLAS dgemm achieves 75% of the theoretical peak on a Kepler GPU. Tasks with little parallelism, significant branching, or that require global synchronization will perform relatively poorly on GPUs.

The Xeon Phi coprocessor is somewhat more flexible than GPU architectures. The current Knights Corner model has up to 60 compute cores (plus one core that is reserved for the operating system), with 4 hyperthreads per core, and an 8 double-precision wide vector unit. It supports traditional CPU thread models such as pthreads and OpenMP. Still, achieving top performance requires all the threads executing instructions on different data using the vector unit. As with the GPU, the Xeon Phi excels at performing dense matrix-matrix operations.

In contrast, CPUs have fewer cores and smaller vector units. They also have more complicated hardware optimizations such as out-of-order execution, branch prediction, and speculative execution. This means that CPUs perform much better at tasks with limited parallelism and significant branching.

Hence, the typical hybrid algorithm splits the overall computation into small tasks to execute on the CPU, and large update tasks to execute on the accelerator [17, 25, 29, 54], as shown in Figure 2a. For instance, in LU and QR factorizations, each step is split into a panel factorization of n_b columns, followed by a trailing matrix update. The panel factorization is assigned to the CPU (red tasks Figure 2a), and includes such decisions as selecting the maximum pivot in each column or computing a Householder reflector for each column. The trailing matrix update is assigned to the accelerator (green tasks in Figure 2a), and involves some form of matrix-matrix multiply. The block size, n_b, can be tuned to adjust the amount of work on the CPU vs. on the accelerator. Optimally, during the trailing matrix update, a look-ahead panel is updated first and sent back to the CPU (green tasks in the *critical path* in Figure 2a). Asynchronous data transfers are used to copy data between the CPU and accelerator while the accelerator continues computing. The CPU performs the next panel factorization while the accelerator continues with the remainder of the trailing matrix update. In this way, the inputs for the next trailing matrix update are ready when the current update finishes. The goal is to keep the accelerator – which has the highest performance – always busy.

This model can be extended to distributed computing, as shown in Figure 2b [30]. Here, a task scheduler assigns the trailing matrix updates, such as the SYRK Level 3 BLAS call, to either the accelerator or the multicore CPU, depending on the availability of resources.

4. Task based programming model

The task based parallel programming model is well established by now and is very successful for DLA, as illustrated with the PLASMA and MAGMA libraries. To provide parallelism, algorithms are split into computational tasks, which in the context of LAPACK algorithms translates to splitting BLAS calls into tasks. Various abstractions can be used to define the model, e.g., as illustrated for QUARK and OpenMP4 in Section 4.1. The resulting algorithms can be viewed as

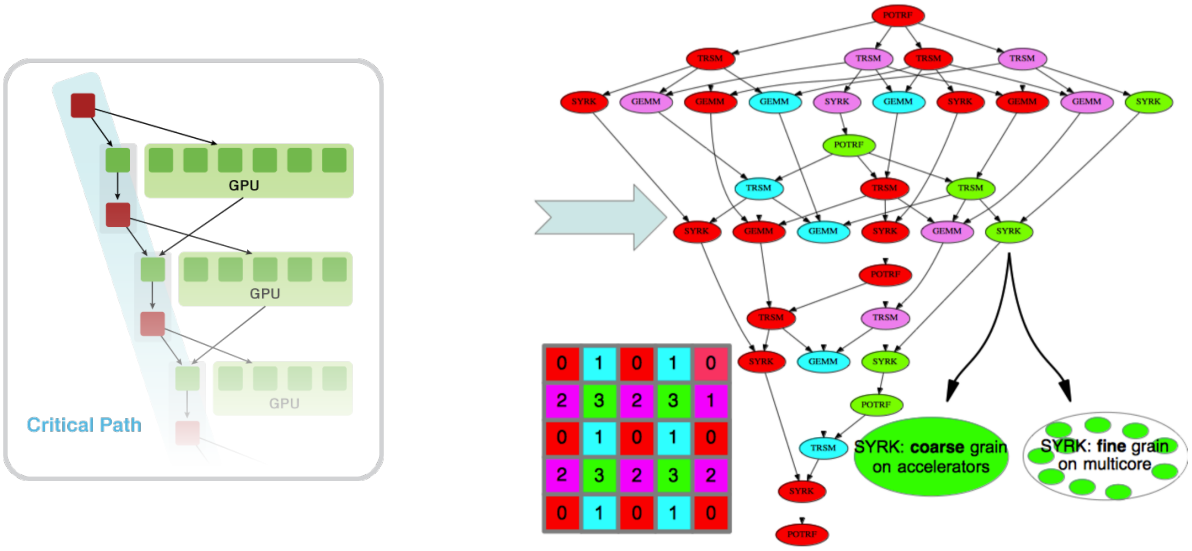


Figure 2. (a) Hybrid algorithm uses a DAG approach where small tasks, typically on the critical path, are scheduled for execution on multicore CPUs, and large data-parallel tasks are scheduled on the accelerator. **(b)** Extension of the hybrid approach to distributed systems using 2D block cyclic layout as in ScaLAPACK, and local/node scheduling of hierarchical tasks for best fit to either multicore CPUs or accelerators

DAGs that can be scheduled for execution on the underlying hardware in various ways, discussed in Sections 4.2, 4.3, and 4.4.

4.1. Algorithms as DAGs

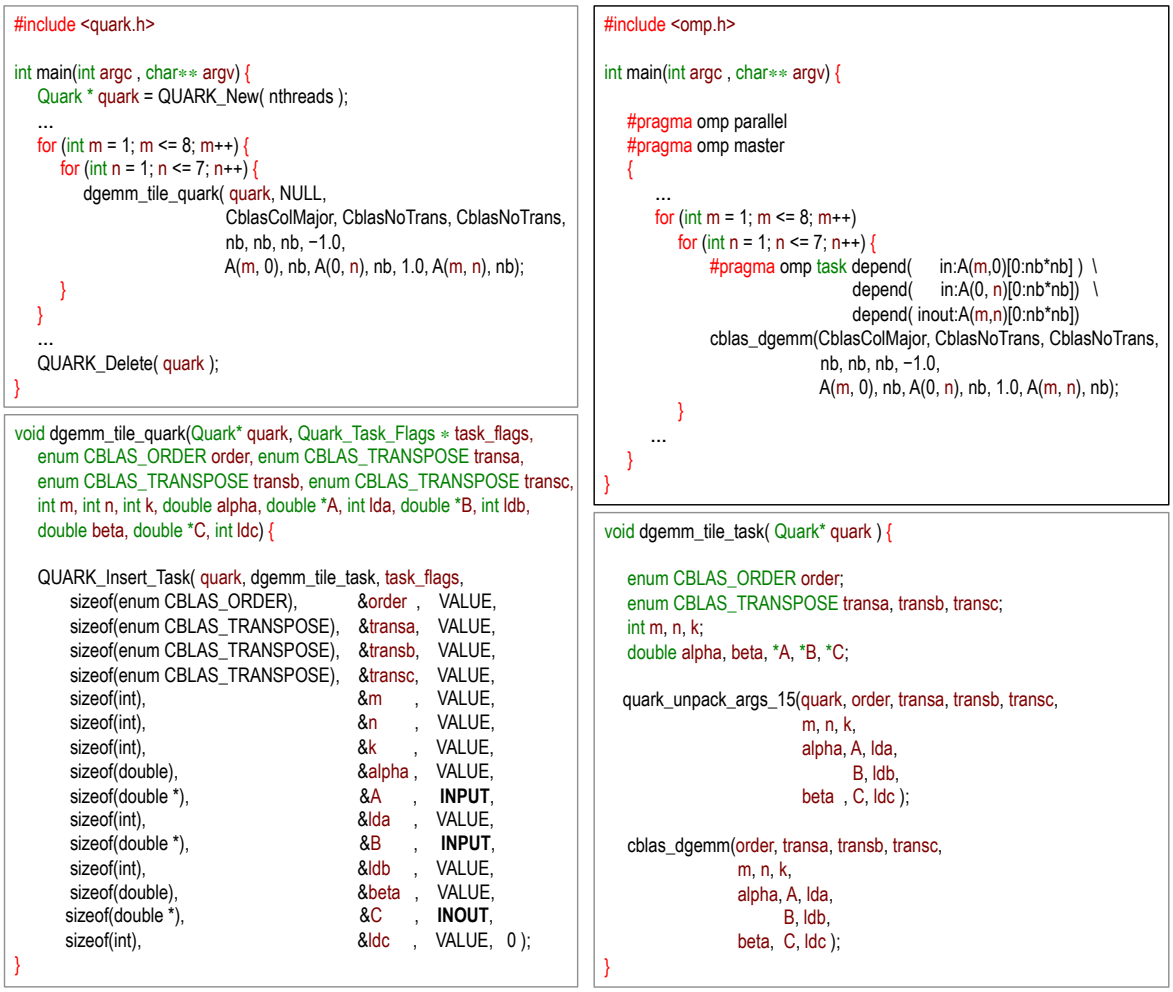
Task-based rank-k update To illustrate the DAG based programming model we present a short example of a general rank-k update (`gemm`) performed by means of two alternative approaches: (i) using the highly customizable runtime QUARK and (ii) applying the standard OpenMP4 runtime. The rank-k update expressed in Eq. (1) is an important building block of many DLA algorithms, including the LU factorization:

$$\mathbf{C} = \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}. \tag{1}$$

Here α and β are scalars, and $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices of sizes $m \times nb$, $nb \times n$, and $m \times n$, respectively. The rank nb is in general the algorithm's blocking size.

Figure 3 shows the example where the matrices are in tile format, and in particular, $\mathbf{A}$ is split into 8 tiles, while B is split into 7 tiles (as shown in Figure 5, Left), each of size $nb \times nb$. Thus, the computation in Eq. (1) is split into 56 tasks. In this case $\alpha = -1$ and $\beta = 1$.

DAG DAG is a well-established concept in the task-based programming world. To construct a DAG, the sequential code is split into basic independent operations, where each operation is performed by means of a new task usually assigned to a processing thread. Each task has data dependencies, specified through its arguments (with clauses like INPUT, INOUT, etc.). Prior to execution, the runtime analyzes data dependencies of each operation and constructs a task graph. The tasks to be performed become DAG nodes and the data dependencies become graph edges. The graph is unfolded on-the-fly at program execution time. Since the task graph



```
#include <quark.h>

int main(int argc , char** argv) {
    Quark * quark = QUARK_New( nthreads );
    ...
    for (int m = 1; m <= 8; m++) {
        for (int n = 1; n <= 7; n++) {
            dgemm_tile_quark( quark, NULL,
                              CblasColMajor, CblasNoTrans, CblasNoTrans,
                              nb, nb, nb, −1.0,
                              A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
        }
    }
    ...
    QUARK_Delete( quark );
}
```

```
void dgemm_tile_quark(Quark* quark, Quark_Task_Flags * task_flags,
    enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE transa,
    enum CBLAS_TRANSPOSE transb, enum CBLAS_TRANSPOSE transc,
    int m, int n, int k, double alpha, double *A, int lda, double *B, int ldb,
    double beta, double *C, int ldc) {

    QUARK_Insert_Task( quark, dgemm_tile_task, task_flags,
        sizeof(enum CBLAS_ORDER),       &order ,   VALUE,
        sizeof(enum CBLAS_TRANSPOSE),   &transa,   VALUE,
        sizeof(enum CBLAS_TRANSPOSE),   &transb,   VALUE,
        sizeof(enum CBLAS_TRANSPOSE),   &transc,   VALUE,
        sizeof(int),                    &m     ,   VALUE,
        sizeof(int),                    &n     ,   VALUE,
        sizeof(int),                    &k     ,   VALUE,
        sizeof(double),                 &alpha ,   VALUE,
        sizeof(double *),               &A     ,   INPUT,
        sizeof(int),                    &lda   ,   VALUE,
        sizeof(double *),               &B     ,   INPUT,
        sizeof(int),                    &ldb   ,   VALUE,
        sizeof(double),                 &beta  ,   VALUE,
        sizeof(double *),               &C     ,   INOUT,
        sizeof(int),                    &ldc   ,   VALUE,   0 );
}
```

```
#include <omp.h>

int main(int argc , char** argv) {

    #pragma omp parallel
    #pragma omp master
    {
        ...
        for (int m = 1; m <= 8; m++)
            for (int n = 1; n <= 7; n++) {
                #pragma omp task depend(     in:A(m,0)[0:nb*nb] ) \
                                 depend(     in:A(0, n)[0:nb*nb])  \
                                 depend( inout:A(m,n)[0:nb*nb])
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                            nb, nb, nb, −1.0,
                            A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
            }
        ...
    }
}
```

```
void dgemm_tile_task( Quark* quark ) {

    enum CBLAS_ORDER order;
    enum CBLAS_TRANSPOSE transa, transb, transc;
    int m, n, k;
    double alpha, beta, *A, *B, *C;

    quark_unpack_args_15(quark, order, transa, transb, transc,
                         m, n, k,
                         alpha, A, lda,
                                B, ldb,
                         beta  , C, ldc );

    cblas_dgemm(order, transa, transb, transc,
                m, n, k,
                alpha, A, lda,
                       B, ldb,
                beta,  C, ldc );
}
```

Figure 3. A rank-k update example of using tasking with QUARK (Left) and OpenMP4 (Right)

might be very large even for a small program the runtime usually applies a window to process and "walk over" a limited amount of nodes. Representing application data as graph edges and computational operations (tasks) as nodes allows for achieving high levels of asynchronicity and parallelism.

QUARK vs OpenMP QUARK (QUeing And Runtime for Kernels) is a runtime engine for dynamic scheduling and execution of applications aimed at multicore/multisocket shared memory systems. This runtime implements a data flow model, where scheduling decisions are made, resolving data dependencies in a task graph. QUARK analyzes and resolves data dependencies at runtime by unfolding the DAG of tasks. QUARK is capable of DAG merging and loop reordering. At scheduling time, the runtime tries to maximize data reuse based on the data locality information. The tasks to be executed are realized by means of a FIFO queue and idling threads are allowed to perform LIFO work stealing. Apart from basic flags presented in Figure 3, QUARK features special flags to control task priorities and set data locality, accumulation, and gathering properties. With QUARK it is also possible to assign multiple tasks to a single thread, as well as switch to manual scheduling of certain tasks.

Compared to QUARK, OpenMP provides a simpler standard interface widely adopted on most modern hardware platforms. Since version 4.0, OpenMP features a `task` pragma with the `depend` keyword making specification of input/output dependencies straightforward.

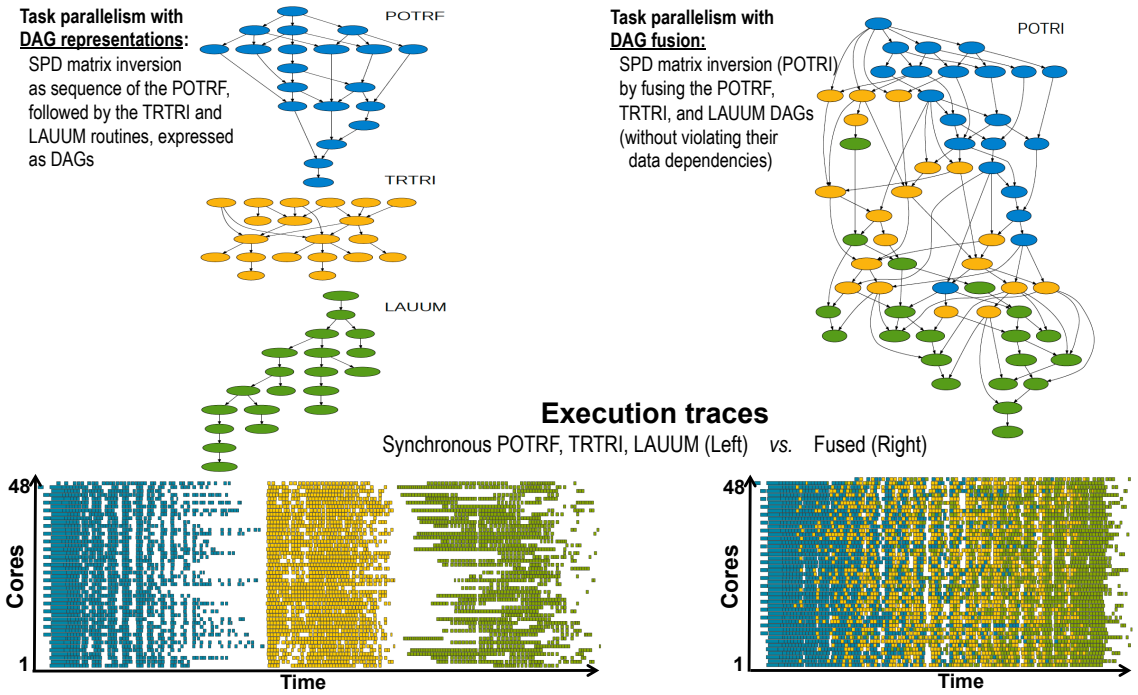


Figure 4. The DAG approach for inverting a symmetric and positive definite (SPD) matrix as a sequence of the POTRF, TRTRI, and LAUUM routines, expressed as DAGs, along with its execution trace on a 48 core CPU system (Left). The same routine, but with fused DAGs, so that data dependencies are not violated, along with its execution trace (Right)

Asynchronous DAG fusion In most execution environments the end of algorithmic step causes a synchronization before the computation can move to the next step of the algorithm. This is very evident in the fork-join implementations of linear algebra algorithms, where synchronization occurs between successive calls to the high-level (BLAS) algorithms. An asynchronous, data-driven runtime environment can dynamically merge multiple algorithms to achieve higher performances. Figure 4 shows the structure of the symmetric positive definite (SPD) inversion algorithm (aka Cholesky inversion) which is implemented using calls to three high-level routines. The algorithm involves three steps: computing the Cholesky factorization ($A = LL^T$), inverting the L factor (computing L^{-1}), and, finally, computing the inverse matrix $A^{-1} = L^{-1^T}L^{-1}$. Using LAPACK, the three steps would be performed by the functions: POTRF, TRTRI, and LAUUM, and would result in a synchronization between each stage. Figure 4 shows the effect of using a dynamic, asynchronous, data-driven execution environment, which can automatically and transparently extract a higher degree of parallelism by fusing the three stages. The data-dependencies between the stages are inferred and tasks from all the stages can be interspersed if the dependencies are fulfilled.

4.2. Static scheduling

For a given DAG, a *static scheduling scheme* passes all the computational tasks to the parallel processing units in a specific execution order before the execution of the DAG. The data movement between the processing units is also statically scheduled. Such a scheme can obtain excellent performance when the programmer knows the properties of the computational tasks and of the target hardware architecture. There have also been extensive studies to automatically tune some of the static parameters in order to improve the performance of the static scheduling.

For example, many of the routines in MAGMA follow a *hybrid programming* model where the computational tasks are statically scheduled on the CPU and GPUs (including multiple GPU and non-GPU-resident factorization algorithms [32, 35, 56, 57]). In this model, BLAS-1 or BLAS-2 based tasks, that tend to be on the critical path, are typically scheduled on the CPU, while the BLAS-3 based tasks are scheduled on the GPU (see Figure 2a). In this fashion, we can take advantage of the data parallelism on the GPU, while the CPU is often efficient at executing the latency-bound tasks. In addition, the scheduling can be designed to respect the data locality while maximizing the overlap of executing the independent tasks on the CPU and GPUs (e.g., look-ahead update of the panel).

One drawback to the static scheduling is that since the tasks are scheduled before the execution starts, the scheduling cannot adapt to changes in the execution environments that may occur during the execution of the program (e.g., the clock frequency and network congestion, or the load imbalance due to the fill-ins during a sparse factorization). Also, in order to obtain the high performance, significant knowledge may be required about the characteristics of the computational tasks and underlying hardware (both of which are becoming extremely complex), and the scheduling needs to be tuned for the specific hardware and potentially for the input data.

4.3. Dynamic scheduling

Some of the shortcomings mentioned for the static scheduling can be overcome by dynamic scheduling heuristics. The general mechanisms of using dynamic scheduling are as given in Section 4.1. The benefit is that a lightweight runtime environment with task superscalar concepts can be used to enable the developer to write serial code while providing parallel execution. In this model, the computation must be split into tasks with specified input dependencies and outputs (as in Section 4.1). Knowledge of the algorithm in terms of the best ways to schedule the execution (e.g., for best fit to the underlying hardware components or to minimize communication, etc.) is not needed in runtimes like StarPU, but if available can be passed to the runtime through task priorities or other mechanisms to help the scheduler to improve performance [14, 25, 32, 52].

Thus, dynamic scheduling can ease the parallel programming efforts and sometimes improve performance (*vs.* static scheduling). Moreover, it can be incorporated in a parallel programming model to unify DLA software for heterogeneous systems with components of various strengths. The challenge here is that in order to efficiently use GPU resources, the workload must have a greater degree of parallelism than a workload designed for multicore CPUs, and conceptually, the Intel Xeon Phi coprocessors are capable of handling workloads somewhere in between the two. To address challenges like these, dynamic scheduling plus task abstractions can be used to develop a unified algorithmic parallel programming model for DLA algorithms capable of fully utilizing a wide variety of heterogeneous resources [26], where it is possible to combine NVIDIA and AMD GPUs, multicore CPUs, and Xeon Phi coprocessors in the same system.

Finally, we highlight that while programming models and scheduling are important, it is the algorithmic design that is of highest significance for performance. In particular, the design should account for reducing communication, as time per flop, network bandwidth (between parallel processors), and network latency are all improving, but at exponentially different rates. So an algorithm that is computation-bound and running close to peak today may be communication-bound in the near future. The same holds for communication between levels of the memory hierarchy. To reiterate what we already stated and stress throughout the paper, related to the

latter, performance is indeed harder to get on new architectures unless hierarchical communications are applied. Hierarchical communications to get top speed now are needed not only for compute bound operations, e.g., BLAS-3, but also for BLAS-2 [36]. Thus, a programming model should provide abstractions for hierarchical tasks that can block computations over the memory hierarchies in order to reduce data movement. This has been demonstrated in DMAGMA [30] for both shared and large scale distributed memory heterogeneous systems. Figure 2b illustrates the approach: a 2D-block cyclic matrix layout is used; a static scheduling between nodes as in ScaLAPACK; DAG algorithm representation; dynamic scheduling within the node; and hierarchical tasks that can be subdivided or grouped when needed.

4.4. Batched scheduling

The purpose of batched scheduling is to deal with workloads that need to solve a large number of relatively small problems that are independent from each other. The size of each individual problem can be so small (e.g., sub-vector/warp in size) that it often fails to provide sufficient parallelism for modern hardware architectures, especially throughput-oriented accelerators. Such workloads arise in many real applications, including astrophysics [41], quantum chemistry [6], metabolic networks [37], CFD and resulting PDEs through direct and multifrontal solvers [60], high-order FEM schemes for hydrodynamics [15], direct-iterative preconditioned solvers [33], and image [43] and signal processing [5].

The development of a multicore CPU-based solution to such workloads can be implemented using the existing software stack. Each individual problem can fit in cache, where existing optimized routines (e.g., as in MKL [34] or ACML [3]) can run efficiently and achieve high performance. Based on our experience [27], a better approach is to assign one core per problem at a time, rather than launching multithreaded kernels on each problem. The former approach achieves better performance if all cores work concurrently on different independent problems. The scheduling of problems among cores can be static or dynamic (e.g., through an OpenMP `parallel for` clause). The latter scheduling achieves better performance in the generic case of having different problem sizes, as problems are assigned dynamically to cores that become idle during computation, in a fashion similar to Single Queue Multiple Servers configuration.

However, the same approach cannot be used for accelerators like GPUs. The caches in such a category of hardware are too small to host even relatively small problems. For example, the L1/shared memory module in a modern NVIDIA GPU can be configured up to 48KB per Streaming Multiprocessor (SM), which cannot host a square matrix of size larger than 78 in double precision. In addition, existing numerical software for accelerators usually focuses on problems with relatively large sizes. For example, matrix factorization algorithms for GPUs [55] are designed in a hybrid fashion to use CPUs for panel factorizations, which lack enough parallelism, and GPUs for trailing matrix updates, which are embarrassingly parallel (mostly `gemm`s). The CPU-GPU communication can be hidden by factorizing a new panel while the update takes place [55]. Unfortunately, such a hybrid approach cannot be used when the matrix is small, since the trailing matrix updates are not big enough to hide the communication of the panel. This is why there is a need to have a new approach for batched workloads.

One of the available solutions can be found in the MAGMA library [2], which uses a fully GPU-based approach. As an example, batched one-sided factorizations are developed using batched BLAS routines. GPU kernels use a 3D grid configuration, where each 2D subgrid is responsible for one problem. The GPU runtime is able to efficiently schedule thread blocks (TBs)

across SMs with relatively large batches. In order to achieve high performance on such small sizes, many kernel optimizations are conducted, such as nested recursive blocking, parallel swapping for batched LU factorization, and using streamed `gemm`s for trailing matrix updates [27]. A common target in designing batched kernels is to maximize the number of TBs that can execute concurrently on the same SM, which eventually increases the throughput of solved problems. In order to achieve this target, the design of batched BLAS kernels tends to reduce the resources required per TB, in terms of shared memory, registers, and even number of threads. This is unlike the common approach used in classical BLAS kernels, where TBs often consume a lot of resources to guarantee high performance.

5. Future directions

5.1. Latest requirements for LA functionalities in applications

The acceleration of contemporary DLA on heterogeneous architectures, and in particular the acceleration of numerical solvers for large linear systems and eigenvalue problems, is still the main concern in many applications. Examples, according to our recent survey among the Sca/LAPACK, PLASMA, and MAGMA users, are: electronic structure calculations, quantum mechanics and chemistry, continuum mechanics, fluid dynamics (spectral methods), aerodynamics and structures, lattice QCD, weather prediction, computational geophysics for electromagnetic data, convex and non-convex continuous optimization, principal component analysis, PCA calculations for data reduction, linear regression, FEM, hierarchical matrix compression, and various sparse solvers and preconditioners. However, besides these application, around 40% of the responders required LA on many independent problems that are of size O(100) and smaller (and sizes up to O(10) for 20% of the cases). These are applications in machine learning, data analysis, signal processing, batched operations for sparse preconditioners, algebraic multigrid, sparse direct multifrontal solvers, QR types of factorizations on small problems, astrophysics, high-order FEM, and others (see Section 4.4).

Another requirement for DLA numerical libraries is to support more general data formats - not just the standard column/row major dense matrices, or the recently introduced tiled matrices, but also user defined, e.g., suitable to describe physical properties featuring multilinear relations, like tensors [1, 8]. Figure 5, Left illustrates the notion of tensor and tensor contraction in DLA, as well as a tensor contraction design using a Domain Specific Embedded Language (or DSEL). Figure 5, Right illustrates the need for tensor contractions in machine learning. The computational characteristics in this case are common to many applications: the operations of interest are in general small, they must be batched for efficiency, and various transformations may have to be explored to transform the batched small computations into regular – and therefore efficient to implement – operations, e.g., `gemm`s [1]. The efforts to provide solutions for these problems must often be multidisciplinary, incorporating LA, languages, code generations and optimizations, domain science, and application-specific numerical algorithms.

5.2. Weaknesses and strengths of current approaches

The tasking approach and the use of BLAS are here to stay. The current state-of-the-art illustrates that these approaches are prevailing, showing advantages in both performance and ease of development. In spite of their strengths, though, they have weakness too. In particular, a

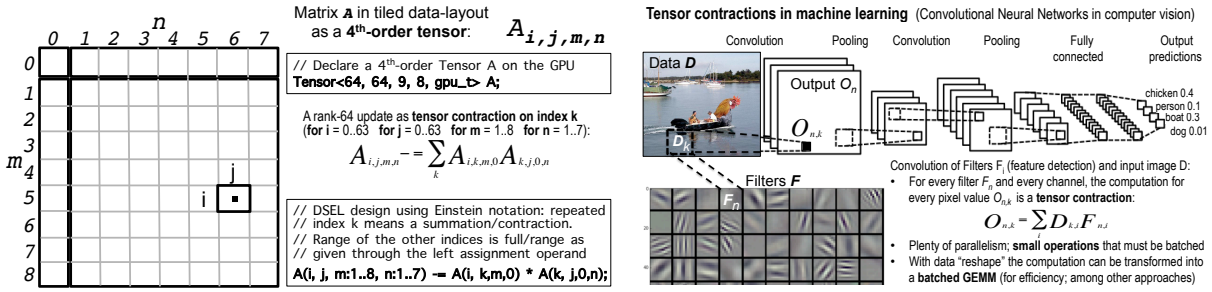


Figure 5. Left: Examples of a 4^{th}-order tensor for tile matrix layout used in DLA, a tensor contraction, and a tensor contraction design using Einstein summation notation through a DSEL. **Right**: Illustration of tensor contractions needed in machine learning

critical weakness is the level of interoperability with other tools and libraries. For example, fusion of DAGs in the tasking model, as in Figure 4, is achievable only within the programing model used for the different DAG components; only at that level does the user/developer of a fused POTRI know the subtasks and their dependencies. If a developer uses third party tools, i.e., does not have access to the subtasks and their dependencies, the fork-join bottleneck between the routines of the libraries used will be encountered. Other weaknesses include: lack of software for small batched problems, and thus lack of support from run-time systems for efficient scheduling of these types of problems; lack of enough flexibility in data structures for data-driven parallel models; deficiencies in mechanisms to deal with heterogeneity in terms of natural enforcement of load balance, especially for distributed systems with different nodes.

5.3. The next generation of parallel programming models for DLA

BLAS, tasking, and scheduling will be indispensable in the next generation of parallel programming models for DLA. Thus, "adapting what we have can (and must) work" [31]. Next generation programming models must address the weaknesses mentioned above. The interoperability issue can be addressed by building models on open standards, e.g., designing runtime systems on top of open standards that support tasking and are competitive with current runtimes. Thus, we advocate the adoption of OpenMP4.5 in the design of parallel programming models for DLA libraries. Load balancing issues can be addressed by better support for hierarchical tasks and data abstractions. Therefore, future directions in MAGMA, for example, include the development of key extensions to OpenACC/OpenMP to access different types of memory hierarchies, the design of data-structure APIs for efficient mapping to extreme-scale heterogeneous systems, and interoperability with other programming models and libraries. Related to new data structures, analysis of current applications of interest – ranging from machine learning to big data analytics – has demonstrated the need to handle higher dimensional data; these are cases where flattening applications to LA on two-dimensional data (matrices) may not be enough. To address these needs, support of tensor data abstractions and batched scheduling in the parallel programming models can be highly beneficial [1].

Conclusions

Support for developing hierarchical algorithms is the main component of current and future parallel programming models for DLA. The evolution in the models follows the expansion of the

memory hierarchies in modern architectures. This has been accomplished historically by the use of BLAS. Now BLAS is used to cover the hierarchies on a single core, as well as many cores on shared memory architectures through parallel BLAS implementations. PBLAS has provided an extension of the BLAS model to distributed systems. On complex heterogeneous nodes, BLAS has been combined with runtime systems, where developers split the computation into tasks (e.g., BLAS-type for a single core), and the runtime then maps the tasks for execution over the machine's components. These models lead to fork-join type approaches where the fork is either parallel BLAS, or other parallel LA algorithms provided by a numerical library (e.g., MAGMA). Future directions will improve interoperability between libraries and tools through the use of widely accepted models like CUDA and open standards like the OpenMP4.5, OpenACC, and/or OpenCL. Thus, new developments will be built and will rely on what we have now. Indeed, for distributed memory systems, we have shown that building on traditional approaches, such as the 2D block cyclic distribution from ScaLAPACK and extensions of heterogeneous models for shared memory systems, works. As a word of caution in using static *vs.* dynamic scheduling, we point out that although dynamic scheduling may have certain benefits in terms of ease of implementation and handling heterogeneity, splitting a regular (DLA) computation into many small tasks and trying to handle dependencies among them can introduce overheads. Remarkably, mapping the HPL benchmark on heterogeneous systems is still based on regular, statically scheduled and executed code. Approaches that use the regularity of DLA must be used, where grouping tasks, as well as the ability to hierarchically split them when needed, can be mapped efficiently to the hierarchical memories of current and upcoming heterogeneous systems.

In summary, future directions for the next generation of parallel programming models for high-performance dense linear algebra libraries on heterogeneous systems must include support of hierarchical tasks, runtimes for hierarchical tasks, batched approaches, new data and task abstractions, as well as hybrid BLAS versions based on them.

This material is based upon work supported by the National Science Foundation under Grants No. ACI-1339822, NVIDIA, Intel, AMD, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

1. A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. High-Performance Tensor Contractions for GPUs. Technical Report UT-EECS-16-738, 01-2016 2016.

2. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

3. ACML - AMD Core Math Library, 2014. Available at `http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml`.

4. E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

5. M. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

6. A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Luc, M. Nooijene, R. Pitzerf, J. Ramanujamg, P. Sadayappanc, and A. Sibiryakovc. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

7. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

8. M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, C. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. Towards a High-Performance Tensor Algebra Package for Accelerators. `http://icl.cs.utk.edu/projectsfiles/magma/pubs/43-smc15_tensor_contractions.pdf`, September 2 2015. Smoky Mountains Computational Sciences and Engineering Conference (SMC'15), Poster, Gatlinburg, TN.

9. R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using smpss. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

11. J. Choi, J. Demmel, I. S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers Design Issues and Performance. *Computer Physics Communications*, 97, aug 1996.

12. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA '95, Lyngby, Denmark, August 21-24, 1995, Proc.*, pages 107–114, 1995.

13. M. Corporation. C++ AMP : Language and programming model, 2012. Version 1.0, August.

14. S. Donfack, S. Tomov, and J. Dongarra. Dynamically balanced synchronization-avoiding lu factorization with multicore and gpus. In *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*, 05-2014 2014.

15. T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

16. J. J. Dongara, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

17. J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pages 1–26, 2014.

18. J. Dongarra, J. Kurzak, P. Luszczek, T. Moore, and S. Tomov. Numerical algorithms and libraries at exascale. `http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/`, October 19 2015. HPCwire.

19. J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

20. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990.

21. P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, Aug. 2012.

22. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OMPSS: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

23. K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, 1st edition, 2012. ISBN-13: 978-0735664739 ISBN-10: 0735664730.

24. F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):17, 2012.

25. A. Haidar, C. Cao, I. Yamazaki, J. Dongarra, M. Gates, P. Luszczek, and S. Tomov. Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads Across Accelerators, Coprocessors, and Multicore Processors. In *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 14)*, New Orleans, LA, 11-2014 2014. IEEE.

26. A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.

27. A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on gpus. *International Journal of High Performance Computing Applications*, 2015.

28. A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations. In *ISC High Performance*, Frankfurt, Germany, 07-2015 2015. Springer, Springer.

29. A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01-2015 2015.

30. A. Haidar, A. YarKhan, C. Cao, P. Luszczek, S. Tomov, and J. Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *17th IEEE International Conference on High Performance Computing and Communications*, Newark, NJ, 08-2015 2015.

31. M. A. Heroux. Exascale programming: Adapting what we have can (and must) work. `http://www.hpcwire.com/2016/01/14/24151/`, January 14 2016. HPCwire.

32. M. Horton, S. Tomov, and J. Dongarra. A class of hybrid LAPACK algorithms for multicore and GPU architectures. In *Proceedings of Symposium for Application Accelerators in High Performance Computing (SAAHPC)*, 2011.

33. E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

34. Intel Math Kernel Library, 2014. Available at `http://software.intel.com/intel-mkl/`.

35. Y. Jia, P. Luszczek, and J. Dongarra. Multi-GPU implementation of LU factorization. In *proceedings of the international conference on computational science (ICCS)*, 2012.

36. K. Kabir, A. Haidar, S. Tomov, and J. Dongarra. On the Design, Development, and Analysis of Optimized Matrix-Vector Multiplication Routines for Coprocessors. In *ISC High Performance 2015*, Frankfurt, Germany, 07-2015 2015.

37. J. L. Khodayari A., A.R. Zomorrodi and C. Maranas. A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic engineering*, 25C:50–62, 2014.

38. Khronos OpenCL Working Group. The opencl specification, version: 1.0 document revision: 48, 2009.

39. J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

40. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.

41. O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

42. S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

43. J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.

44. R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.

45. C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, J. Dongarra, H. Anzt, M. Gates, A. Haidar, Y. Jia, K. Kabir, I. Yamazaki, and J. Labarta. Heterogeneous streaming. In *IPDPSW, AsHES 2016 (accepted)*, Chicago, IL, USA, May 23 2016.

46. OpenACC Non-Profit Corporation. The OpenACC application programming interface version 2.0. `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`, June 2013.

47. OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

48. OpenMP Architecture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

49. OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov 2015.

50. J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta. Cellss: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

51. A. Pop and A. Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4), 2013.

52. F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *ICS*, pages 365–376, 2012.

53. M. Tillenius. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

54. S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010. http://dx.doi.org/10.1016/j.parco.2009.12.005, DOI: 10.1016/j.parco.2009.12.005.

55. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

56. I. Yamazaki, S. Tomov, and J. Dongarra. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. In *proceedings of the international conference on computational science (ICCS)*, 2012.

57. I. Yamazaki, S. Tomov, and J. Dongarra. Non-GPU-resident symmetric indefinite factorization. *Submitted to ACM Transactions on Mathematical Software (TOMS)*, 2016.

58. Y. Yan, B. M. Chapman, and M. Wong. A comparison of heterogeneous and manycore programming models. http://www.hpcwire.com/2015/03/02/a-comparison-of-heterogeneous-and-manycore-programming-models, March 2 2015. HPCwire.

59. A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, 2012.

60. S. N. Yeralan, T. A. Davis, and S. Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.