Online Methods

Application overview

The core parallelization routines in sPEGG are implemented using Thrust (Hoberock and Bell 2010), a highly optimized parallel algorithms library loosely resembling the widely used C++ standard template library (STL) (Stepanov and Lee 1994). To exploit GPGPU hardware, the present generation of Thrust uses CUDA (Nickolls et al. 2008). Thus, the GPGPU version of sPEGG can currently only run on CUDA-enabled NVIDIA hardware, although this may change as Thrust expands to support other GPGPU programming languages (e.g., OpenCL). Thrust also allows us to use multi-core CPU processors for parallellization. Currently, most consumer desktops have only a small number (generally 2-8) of CPU cores (e.g., Hruska 2012). However, if CPU core counts increase dramatically at affordable prices while GPGPU hardware improvements stagnate, sPEGG's use of the Thrust library enables it to be well-positioned to exploit such hardware developments.

In general, sPEGG relies on an object-oriented design (e.g., Booch 1982, Rumbaugh et al. 1990 and Martin 2002). For instance, sPEGG models each species as a separate object, consisting of thrust vectors stored on GPGPU memory that describe the phenotypes and genotypes of individuals. The calculations are then performed in parallel on individual-level data stored in the species objects. Models created with sPEGG can then be written in standard C++. Supplementary Figure S4 illustrates how the species objects of sPEGG could be used in an IBM.

sPEGG is released under the GNU Public License v3 (Stallman 2007). The most recent version of sPEGG, further documentation, source code for the case studies (including their parameter values and initial conditions) and a short tutorial are available at `https://github.com/kewok/spegg`.

Approach to GPGPU parallelization

The simulation of dynamical models in ecology and evolution is inherently serial (as the state variables are often Markovian). Thus, parallelization in sPEGG occurs within a time-step of the model across individuals. For many modeling problems, the benefits of parallelization, particularly on GPGPUs, are often greatest when the number of calculations are large (e.g., Harish and Narayanan 2007, Nyland et al. 2007, and Owens et al. 2008). Hence, the performance gains due to sPEGG are largest when calculations within each time-step can be carried out over a large number (typically $> 10^6$) of individuals. This may present a potential problem for accelerating simulations that model the eco-evolutionary dynamics of a smaller number of individuals, as can often be the case in some systems (e.g., Pelletier et al. 2009).

A single simulation run of a given model may only simulate a small number of individuals - for instance, on the order of hundreds to tens of thousands, which are well below abundances at which the benefits of parallelization on the GPGPU may be apparent. However, sPEGG's power lies in its ability to simulate a large number of individuals across several simulation runs simultaneously. Consider simulating a model of 1000 individuals across 10,000 different parameter combinations. This exercise requires performing calculations on 10^7 individuals, which is well within the range of the number of calculations whose performance can benefit from parallelization on a GPGPU.

To accomplish this, sPEGG organizes individuals into discrete patches. The population of individuals within a given patch is referred to as a deme. Individuals can potentially migrate between patches, although the user can also prohibit migration to and from patches. In sPEGG, patches that are not linked by migration can therefore be thought of as representing independent, replicate simulations. For instance, in the example above, sPEGG will simulate a model of 10,000 patches consisting of 1000 individuals each, where the model's dynamics within each patch are governed by a unique parameter combination. Supplementary Figure S5 illustrates an example of how sPEGG uses patches to enable parallel calculations to be performed simultaneously for all individuals. A given patch can contain multiple species, and, between migration events (if they are allowed), individuals only interact with other individuals within the same patch.

Customizing sPEGG

As noted in the main text, the sPEGG code-base for simulating a species-specific model requires some level of user customization. Functions initializing the data, simulating mating and reproduction, and updating the trait values/phenotypes of individuals are completely general and easily customizable for researchers using models to address a variety of questions. sPEGG includes built-in alternative routines and classes enabling researchers to readily implement code for diverse systems, such as different mating systems and inheritance mechanisms (see the main text for details).

For situations that are harder to generalize, such as the rules describing how individual phenotypes are updated in response to heterospecific individuals during each time step, sPEGG provides methods, as well as readily usable classes, to facilitate the updating of phenotypes. The case studies illustrate the use of these classes for performing calculations between individuals of different species as well as for modeling resource dynamics and individual consumption behavior. Finally, the individual-based models used in the case studies can be explored further by merely changing the numerical values in the corresponding configuration (text) files.

Comparison Between Serial and Parallel Versions of the Case Studies

To assess the performance advantages of using sPEGG in each of the case studies in the main text, each IBM was re-coded by hand using equivalent C++ classes, iterators and functions (invoking functions from the widely-used GNU scientific library - Galassi et al. 2007 - which relies on inherently serial algorithms) for the functions, classes, and model-specific code in sPEGG. We then applied optimization techniques, which are known to improve execution time in a serial context, to the resulting code base. To compare the serial and parallel versions of our models, we simulated our serial IBMs on a single 3.6 Ghz (Intel core i7 3820) CPU processing core and used one half of an NVIDIA GTX 690 GPU (restricting program access to 2GB of GPU RAM and 1536 CUDA cores) to assess the parallel version's performance. All optimizing transformations provided by the compilers (g++ and nvcc) were enabled.

References

Booch, G. 1982. Object-oriented design. *ACM SIGAda Ada Letters* 1:64–76.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., and Rossi, F. 2007. GNU Scientific Library Reference Manual, 3rd Ed. Network Theory.

Harish, P. and Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. *High Performance ComputingHiPC 2007* 4873:197–208.

Hoberock, J. and Bell, N. 2010. Thrust: A Parallel Template Library.

Hruska, J. 2012. The death of CPU scaling: From one core to many and why were still stuck. *ExtremeTech* .

Martin, R. C. 2002. Agile Software Development, Principles, Patterns, and Practices. Alan Apt Series. Prentice Hall.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with CUDA. *Queue* 6:40–53.

Nyland, L., Harris, M., and Prins, J. 2007. Fast n-body simulation with {CUDA}. *GPU gems* 3:677–695.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. 2008. GPU Computing. *Proceedings of the IEEE* 96:879–899.

Pelletier, F., Garant, D., and Hendry, A. 2009. Eco-evolutionary dynamics. *Philosophical Transactions of the Royal Society B: Biological Sciences* 364:1483–1489.

Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W. 1990. Object-oriented modeling and design. Prentice-Hall.

Stallman, R. 2007. GNU General Public License v3.

Stepanov, A. and Lee, M. 1994. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project.