Auto-Generation of Parallel Finite-Differencing Code for MPI, TBB and CUDA

D. P. Playne
Computer Science, IIMS
Massey University
Auckland, New Zealand
d.p.playne@massey.ac.nz

K. A. Hawick
Computer Science, IIMS
Massey University
Auckland, New Zealand
k.a.hawick@massey.ac.nz

***Abstract*—Finite-difference methods can be useful for solving certain partial differential equations (PDEs) in the time domain. Compiler technologies can be used to parse an application domain specific representation of these PDEs and build an abstract representation of both the equation and the desired solver. This abstract representation can be used to generate a language-specific implementation. We show how this framework can be used to generate software for several parallel platforms: Message Passing Interface (MPI), Threading Building Blocks(TBB) and Compute Unified Device Architecture(CUDA). We present performance data of the automatically-generated parallel code and discuss the implications of the generator in terms of code portability, development time and maintainability.**

***Keywords*-automatic code generation; MPI; TBB; CUDA; accelerators; portability; multi-platform.**

I. Introduction

Finite-difference methods are a well known technique for solving partial differential equations [1]. Although finite-element and other matrix-formulated methods are very popular for irregular mesh problems [2], finite-difference methods continue to find use in computational simulations and generally are straightforward to parallelise using geometric stencil methods of decomposition which attain good computational speedup [3]–[5]. Although finite-difference methods are quite feasible to hand-parallelise for low-order stencils when a small number of neighbouring cells is required for each calculation, in cases when higher-order calculus operations are employed [6] the codes become: very complex; hard to implement manually; and very difficult to verify since a small programming error concerning a data index may still lead to a numerically plausible solution that is hard to spot as being wrong. It is therefore very attractive to be able to employ automatic code generation [7] to this problem [8], [9].

In this paper we address two key issues associated with finite-difference methods on parallel computing. The first is the use of auto code-generation methods to specify partial differential equations as a high-level problem using calculus terminology and involves building a software tool-set to generate the programming language source-code to implement the solvers. The second concerns the long-standing problem of portability across parallel programming platforms.

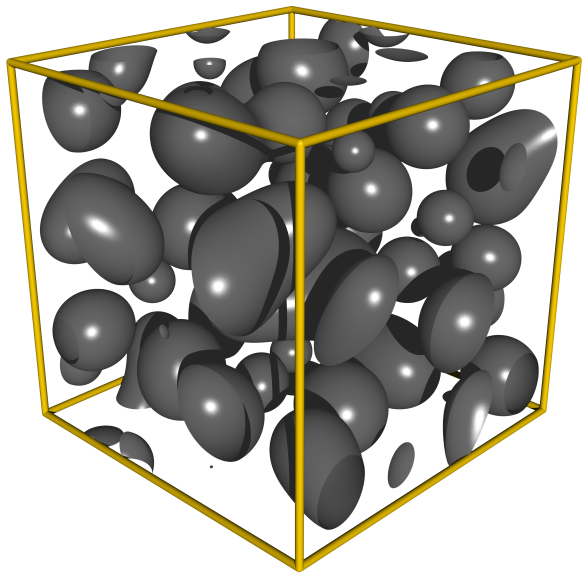

Figure 1. An example three-dimensional Cahn-Hilliard system with a size of 256^3. Ray-traced rendered from a simulation using automatically generated code.

At the time of writing parallel computing is once again attracting great interest and attention as the world faces up to the problems of power consumption of CPUs and the non-continuance of the previous development trend [10] in CPU clock frequencies which used to double approximately every eighteen months in accordance with Moore's law [11]. It is possible to bring parallelism to bear on many problems using hybrids of cluster-computing approaches; accelerator technologies such as general purpose graphical processing unit (GP-GPU); and the use of many threads within a conventional multi-core CPU. These are typified by software technologies such as the open standard Message Passing Interface (MPI) [12], [13]; NVIDIA's Compute Unified Device Architecture (CUDA) [14]–[16] for GPUs; and Intel's Thread Building Blocks (TBB) [17], [18] software for multi-threaded programming multi-core devices, respectively. It is however tedious, error prone and non-

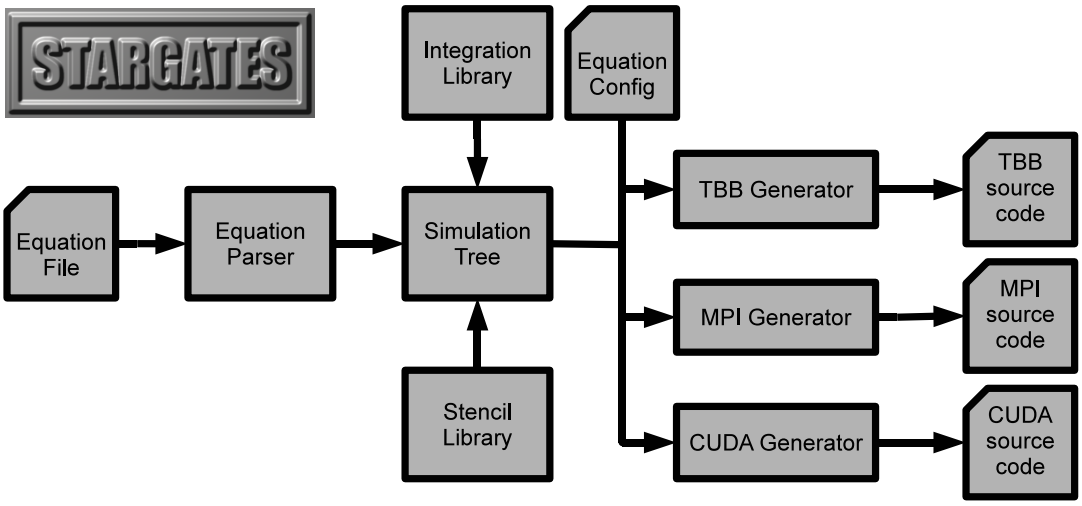

Figure 2. The architecture of the STARGATES generator. Shows logical steps of how an Equation is parsed into a Simulation Tree with input from the Integration and Stencil Libraries and subsequently explored by an Output Generator to produce target code output.

trivial for a programmer or even a programming team to implement an application that works well across all these three parallel paradigms or platforms – even for a problems like finite-difference equation solving that have relatively well known solutions.

We have found that conceptually a great deal is shared between hand-written applications for these platforms and have been able to distill some of the commonalities out. We have developed an automatic code generation system based on compiler technologies such as ANTLR [19] and our own code-generation engines. The use of automatic code generation techniques for numerical solvers is not new [20], but we have been able to go beyond the skeletal and template approaches [21] that have had success in the past. This system allows us to pose a partial differential equation problem in calculus-like terminology and to generate efficient serial or parallel code for a finite-difference solver. The parallel platform that the actual C/C++ [22] compatible output code is generated for can be: MPI; TBB; or CUDA, which we discuss in this present paper, or other platforms such as OpenMP [23], pThreads [24], and other open-standard or proprietary systems.

Our contributions in this paper are therefore: to show how a problem like finite-differencing can be semi-automated to produce correct and efficient parallel source code for use on MPI, TBB and CUDA; but also to show how this tool approach allows exploration of much more sophisticated and complex algorithms and partial differential equations than would otherwise be possible either with manually generated parallel code let alone with a serial code.

In this paper we discuss and demonstrate our system using as an application example a second-order-space/second-order-time partial differential equation – the Cahn-Hilliard field equation [25]–[27] for phase separation in materials science (See Figure 1). Our tools and methods apply to much more complex and elaborate examples such as partial differential equations in complex fluid flow; plasma dynamics; gravitational field theory and superconductors [28]–[30] – and in which areas we are also working. In this paper however, we focus on the parallel computing platform generation aspects and although we do present performance analysis results as proof of the value-add of our approach, we do not dwell on the physics and other application domain-specific details of the problem.

This article is structured as follows. In section II we discuss the conceptual framework of our automatic code generator and the details of the individual TBB, MPI and CUDA code generators. In section III we present fragments of the generated code and their performance results. Finally in section IV we discuss the implications of the generator on parallel code portability, maintainability and performance.

II. STARGATES Architecture

Our simulation generator architecture is named STARGATES or "Simulation Targeted Automatic Reconfigurable Generator of Abstract Tree Equation Systems". This architecture parses a mathematical description of a partial-differential field equation and generates a finite-difference simulation of the equation. The concept behind this generator is to separate the simulation from the implementation. The equation is parsed by an ANTLR [19] equation parser and together with integration and finite-difference stencil libraries can construct an abstract tree representation of the entire simulation. This abstract representation can then processed by output generators to create output code. The architecture of STARGATES can be seen in Figure 2.

One benefit of this architecture is the ease with which simulation code can be maintained. Because the equation descriptions are not tied to any architecture, dimensionality or integration method, only one equation definition needs to be maintained to produce a large number of different implementations. Simulation implementations can be easily generated for any number of dimensions by using the Stencil Library and for any integration method in the Integration Library.

Another advantage of this architecture is the distinction between the simulation description and the details of the parallel implementation. All of the target specific information is defined by the output generator which allows a new target architecture to be easily included with no change to the rest of the system. Because the individual generators have full control over the output code, it allows a large degree of freedom to implement different parallel decomposition algorithms and heuristics for applying appropriate optimisations depending on the properties of the simulation. These output generators can be as complex as the developer wishes to make them.

In this paper we specifically discuss how output generators can be easily constructed to portably generate **parallel** code for three of the most popular (and different) parallel architectures: Multi-Core processors using TBB, Cluster Nodes with MPI and Graphical Processing Units using CUDA.

A. Cahn-Hilliard Equation Example

We use the Cahn-Hilliard field equation as our example in this paper. The Cahn-Hilliard equation in it's simple form is shown in Equation 1. For a complete derivation of this equation see [26].

$$\frac{\partial u}{\partial t} = \mathrm{M}\nabla^2\left(-\mathrm{B}u + \mathrm{U}u^3 - \mathrm{K}\nabla^2 u\right) \tag{1}$$

We write this equation in the following ASCII textual form so it can be parsed by the STARGATES equation parser which is implemented in ANTLR:

```
float M;
float B;
float U;
float K;
float[] u;
d/dt u = M * Laplacian{(-B*u + U*(u*u*u) -
         K * Laplacian{u})};
```

This representation defines both the equation and the data types of the variables and fields. When STARGATES parses this equation description it will generate a tree representation of the equation which can then be traversed by the output generators to create the simulation code. When the equation is parsed, any nested stencil operators (in this equation the Laplacian operator) will be combined using the Stencil Library. This equation contains two nested Laplacian operators which will be combined appropriately. A more complete discussion on how finite-differencing stencils are combined can be found in [31]. The tree representation of this equation with combined operators is shown in Figure 3.

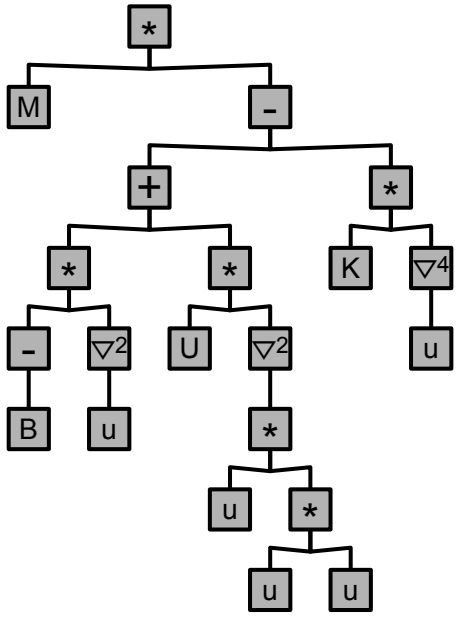

Figure 3. A tree representation of the Cahn-Hilliard equation produced by our ANTLR equation parser.

This equation tree can be traversed by the output generators to produce the code to compute the simulation. In this case all three examples use C syntax and we show the code to compute the equation for a single lattice cell in Listing 1. However it should be noted that the system is not limited to using C syntax, output generators can quite easily generate simulation code in any target language.

Listing 1. The equation calculation code generated by traversing the Cahn-Hilliard cook equation tree. Example shown in C syntax.

```
M*(
(-B)*(                  (unym1x) +
            (unyxm1) + (-4*unyx) + (unyxp1) +
                       (unyp1x))+
U*(             (unym1x*unym1x*unym1x) +
                (unyxm1*unyxm1*unyxm1) +
                  (-4*unyx*unyx*unyx)  +
                (unyxp1*unyxp1*unyxp1) +
                (unyp1x*unyp1x*unyp1x))-
K*(                      (unym2x) +
        (2*unym1xm1)+(-8*unym1x)+(2*unym1xp1) +
(unyxm2)+(-8*unyxm1)+(20*unyx)+(-8*unyxp1)+(unyxp2)
        +(2*unyp1xm1)+(-8*unyp1x)+(2*unyp1xp1) +
                       (unyp2x)))
```

B. Generating TBB code

Intel's Threading Building Blocks allows applications to be easily parallelised for computation on multi-core processors. TBB provides a library of methods and common parallel constructs that developers can use to perform elements of computation in parallel without having to explicitly

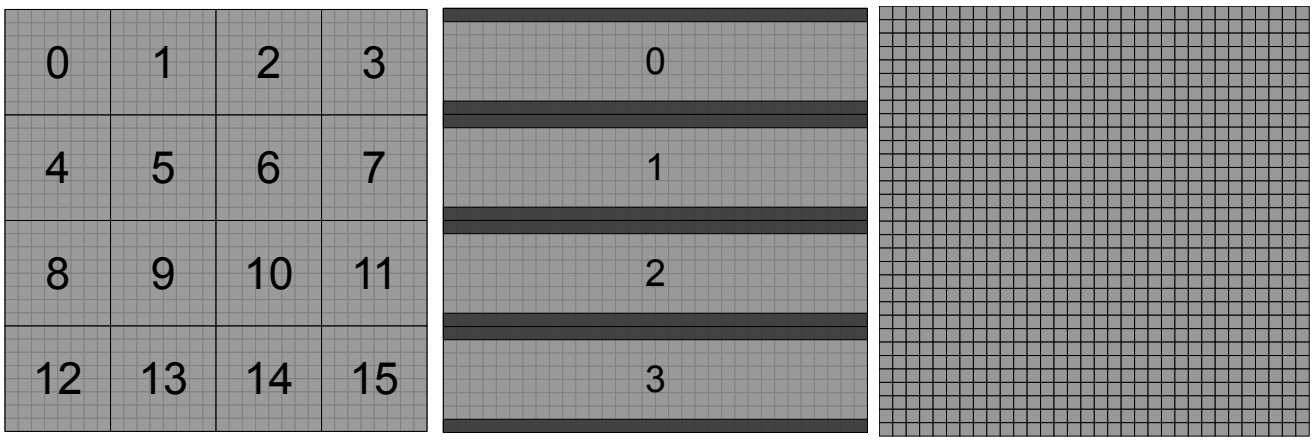

Figure 4. Methods of parallel decomposition for TBB (left), MPI (centre) and CUDA (right). TBB partitions the field into compute blocks (numbered 0-15) which will be distributed one at a time to the available processor cores. MPI splits the field in the highest dimension to separate the computation between available nodes (in this case four nodes), the dark cells represent the borders that must be communicated between nodes after each time-step. CUDA creates one thread to perform the computation for each cell in the lattice, effectively partitioning the computation with the same resolution as the lattice itself.

deal with threads and sychronisation between them. The main advantages TBB provides over more low level systems such as pThreads [24] is the ease of use simplicity of the code. The TBB output generator we have written uses the `parallel_for` to iterate over the lattice and perform the computation in parallel.

Algorithm 1 Pseudo code for generating a TBB finite-differencing solver. This generator creates one function for each integration step and uses the `parallel_for` function to iterate over the cells in parallel.

```
generate includes
generate parameters

for all steps in integration method do
  generate TBB wrapper
  generate TBB iteration code
  generate neighbour access code
  traverse equation tree to generate code
end for

MAIN
generate TBB initialisation
generate equation variables
generate integration method variables
generate time step iteration code
for all steps in integration method do
  generate TBB parallel_for call
end for
```

`parallel_for` will call an update function on each block of data (determined by a `block_range` structure) on any available core until all the computation has been completed. The user can control the resolution with which the field is decomposed into compute blocks. The optimal size of compute block will depend on the simulation. In general if the number of compute blocks is significantly larger than the number of available cores then TBB will be able to perform some automatic load balancing as it will continue to distribute compute block to any idle core. This method of block decomposition can be seen in Figure 4.

Algorithm 1 shows the process that the TBB output generator performs to produce the output TBB code. Most of these steps will depend on the parameters of the simulation being generated. For example the iteration code will depend on the dimensionality of the simulation being generated, these simulation parameters are defined in the configuration file.

When a two-dimensional simulation is being generated, the field will be partitioned into blocks using the `blocked_range2d` structure. Simulations any other dimensionality will use the one-dimensional `blocked_range` structure. This is one example of how specific optimisations can be included in the output generators. The main components of the generated TBB code is shown in Listing 2 in Section III.

C. Generating MPI code

MPI is the most commonly used communication interface for cluster or distributed computers [12]. MPI defines methods for synchronising nodes and sending/receiving data between distributed nodes. For such a system each node will store a section of the field and update it's part of the field. The main challenge of decomposing a finite-differencing simulation across such a system is managing the border communication between the nodes.

Because the computation of a cell depends on the neighbouring values, after each step the cell information on the border of the field split must be communicated to the neighbouring nodes. There are many ways of splitting the field across nodes on a cluster and each has it's individual benefits. The code produced by this generator splits the field evenly in the highest dimension as shown in Figure 4. The high level algorithm of the MPI output generator is shown Algorithm 2.

Algorithm 2 Pseudo code for generating an MPI finite-differencing solver. Generates a MPI program with a single update function that performs all the necessary steps of the integration method and communicates the bordering cells between each step.

```
generate includes
generate parameters

generate MPI function
for all steps in integration method do
  generate iteration code
  generate neighbour access code
  traverse equation tree to generate code
  generate MPI communication code
end for

MAIN
generate MPI initialisation
generate equation variables
generate integration method variables
generate time step iteration code
generate MPI function call
```

This MPI generator decomposes the field evenly between the hosts in the highest dimension and communicates the border information (as determined by the width of the equation stencils) using MPI. This is only one of many possible ways of decomposing the field between nodes but has been chosen as a proof of concept. Listing 3 in Section III shows the main features of the generated MPI code.

D. Generating CUDA code

In recent years GPUs have emerged as a very popular data-parallel computing architecture. NVIDIA's CUDA has proven itself to be the API of choice offering the best ease of use and widest acceptance. Rather than traditional methods of splitting computation into a number of units comparable to the number of processors, CUDA programs split computation into the smallest unit of work. Each unit of work or kernel in a CUDA program performs the finite-difference update on one single lattice cell. This method of decomposition is shown in Figure 4 and the important code to compute a finite-difference simulation can be seen in Listing 4 in Section III.

Algorithm 3 Pseudo code for generating a CUDA finite-differencing solver. This generator creates one function for each integration step.

```
generate includes
generate parameters

for all steps in integration method do
  generate CUDA function
  generate thread id calculation
  generate neighbour access code
  traverse equation tree to generate code
end for

MAIN
generate CUDA initialisation
generate equation variables
generate integration method variables
generate device memory initialisation
generate copy data from host to device
generate CUDA run-time parameters
generate time step iteration code
for all steps in integration method do
  generate CUDA call
end for
generate copy data from device to host
```

Because GPU devices have their own memory, all initial data must be copied into the device and the results copied back to the host afterwards. This does not represent a significant overhead because no memory copies are necessary during the computation of the simulation. These data copies are shown in the CUDA generator algorithm (See Algorithm 3).

One major performance consideration for CUDA applications is the type of memory used. This was very important in previous generations of GPU as the had no automatic cache available whereas the current generation Fermi cards automatically cache access to global memory. Generators can be easily written to make use of different memory types based on the target GPU architecture.

III. Generated Code and Performance Results

The main results of this this work is STARGATES ability to generate correct and efficient parallel code from a simple equation description. We present code-segments from the generated code produced by STARGATES for TBB, MPI and CUDA (see Listings 2, 3 and 4). These code fragments show the lattice allocation, main time-step loop and parallel iteration code.

The TBB code in Listing 2 shows the important fragments for the simulation. The lattices necessary for the integration

method (this example uses the Euler method for simplicity) are allocated on the host using `new`. For each time step the update function is called using `parallel_for` and the update function will iterate through the compute block defined by the `blocked_range2d` parameter. For each lattice cell, the equation will be computed using the code shown in Listing 1.

Listing 2. TBB code automatically generated by STARGATES for a finite-differencing simulation using Euler integration in two-dimensions. Uses `parallel_for` to call `operator` for each compute block.

```
void operator() (const blocked_range2d<int> &r) {
  int yb = r.rows().begin();
  int ye = r.rows().end();
  int xb = r.cols().begin();
  int xe = r.cols().end();
  for(int iy = yb; iy != ye; iy++) {
    for(int ix = xb; ix != xe; ix++) {
      //compute equation for cell ix,iy
    }
}

int main() {
  task_scheduler_init init;
  ...
  float *un = new float[X*Y];
  float *unh = new float[X*Y];
  for(int t = 0; t < 1024; t++) {
    parallel_for(blocked_range2d<int>(
                 0, Y, Y/16, 0, X, X/16),
                 euler(un,unh,h));
    swap(&un, &unh);
  }
  ...
}
```

Listing 3 shows the important code for the MPI implementation of the Cahn-Hilliard simulation. The main function contains code that initialises MPI and identifies the node's id and neighbours. The memory for the lattices are allocated for each node store the section of the field that the node is responsible for as well as the extra bordering cells necessary to compute the simulation. In the update function each node will compute the equation for its section of the field and communicate the borders to its neighbours using MPI_Isend and MPI_Irecv.

Listing 3. Generated MPI code for a two-dimensional finite-differencing simulation using Euler integration. `euler` performs a single computation step for each node's field and communicates the borders to the neighbouring nodes.

```
void euler(float *un,float *unh,float h) {
  for(int iy = HALO; iy < Y+HALO; iy++) {
    for(int ix = 0; ix < X; ix++) {
      //compute equation for cell ix,iy
    }
  }
  MPI_Isend(&um[HALO*X], HALO*X,
            MPI_FLOAT, idm1, 0,
            MPI_COMM_WORLD, &sendm1);
  MPI_Irecv(&um[(Y*X) + (HALO*X)], HALO*X,
            MPI_FLOAT, idp1, 0,
            MPI_COMM_WORLD, &recvp1);
  MPI_Isend(&um[Y*X], HALO*X,
            MPI_FLOAT, idp1, 0,
            MPI_COMM_WORLD, &sendp1);
  MPI_Irecv(&um[0], HALO*X,
            MPI_FLOAT, idm1, 0,
            MPI_COMM_WORLD, &recvm1);
  MPI_Wait(&recvm1, MPI_STATUS_IGNORE);
  MPI_Wait(&recvp1, MPI_STATUS_IGNORE);
  MPI_Wait(&sendm1, MPI_STATUS_IGNORE);
  MPI_Wait(&sendp1, MPI_STATUS_IGNORE);
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);int num_hosts;
  MPI_Comm_size(MPI_COMM_WORLD, &num_hosts);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  idm1 = (id == 0) ? num_hosts-1 : id - 1;
  idp1 = (id == num_hosts-1) ? 0 : id + 1;
  MPI_Request sendm1;
  MPI_Request sendp1;
  MPI_Request recvm1;
  MPI_Request recvp1;
  ...
  float *un = new float[((HALO*2)+Y) * X];
  float *unh = new float[((HALO*2)+Y) * X];
  for(int t = 0; t < 1024; t++) {
    euler(un,unh,h);
    swap(&un, &unh);
  }
  MPI_Finalize();
}
```

The main code sections of the CUDA implementation is shown in Listing 4. In the main function the memory for the lattices are allocated on the device with `cudaMalloc` and the data is copied in and out of the device using `cudaMemcpy`. One thread is created for each lattice cell and performs the update function `euler` which will calculate that thread's id and compute the equation for that cell using the equation code given in Listing 1.

Listing 4. Generated two-dimensional Cahn-Hilliard simulation using Euler integration in CUDA. `block` and `grid` are parameter to control the threads to compute the simulation and `euler` is the function that each thread will perform.

```
_global_ void euler(float *un,float *unh,float h) {
  int ix = (blockIdx.x*blockDim.x) + threadIdx.x;
  int iy = (blockIdx.y*blockDim.y) + threadIdx.y;
  //compute equation for cell ix,iy
}
int main() {
  cudaSetDevice(0);
  ...
  float *un, *unh;
  cudaMalloc((void**) &un,  X*Y*sizeof(float));
  cudaMalloc((void**) &unh, X*Y*sizeof(float));
  cudaMemcpy(un, u, X*Y*sizeof(float),
             cudaMemcpyHostToDevice);
  cudaMemcpy(unh, u, X*Y*sizeof(float),
             cudaMemcpyHostToDevice);
  dim3 block(32,32);
  dim3 grid(Y/block.y, X/block.x);
  for(int t = 0; t < 1024; t++) {
    euler<<< grid, block >>>(un,unh,h);
    cudaThreadSynchronize();
    swap(&un, &unh);
  }
  cudaMemcpy(u, un, X*Y*sizeof(float),
             cudaMemcpyDeviceToHost);
}
```

Automatically generating parallel simulation code is only

Table I
SPEEDUP FACTORS OF AUTOMATICALLY GENERATED PARALLEL IMPLEMENTATIONS OVER A SINGLE CORE CPU IMPLEMENTATION. DATA ACCURATE TO ONE DECIMAL PLACE.

Architecture	Speedup
C++ (3.33GHz Intel i7-980X)	1.0
TBB (3.33GHz Intel i7-980X)	6.6
MPI (2x 2.66GHz Intel Core2 Quad Q9400)	1.8
MPI (4x 2.66GHz Intel Core2 Quad Q9400)	3.7
MPI (8x 2.66GHz Intel Core2 Quad Q9400)	7.3
CUDA (NVIDIA GTX480)	97.9

useful if the generated simulations offer a useful performance benefit. To give an idea of the performance of the generated simulations we have compared them to a hand-written single-core CPU implementation of the simulation. This implementation has been compared to the generated TBB implementation on a six-core Intel i7 980X, the MPI implementation on 2, 4 and 8 Intel Core2 Quad Q9400 nodes in a cluster and the CUDA implementation on a NVIDIA GeForce GTX480. These performance results are shown in Table I.

All of the generated parallel implementations provide a useful speed-up over the single-core implementation. Most notably the CUDA implementation which provides a speed-up of almost 100x.

IV. SUMMARY & CONCLUSIONS

We have shown that human-readable source code for finite-differencing simulations can be automatically generated from a simple ASCII textual representations of the calculus of a partial differential equation. The method of creating an abstract simulation representation allows the source code generation to be completely separated from the equation specification. This allows simulation source code for widely different parallel architectures to be generated from the same equation representation. This generator structure also allows dimension and architecture specific optimisations to be included in the code-generation while also maintaing the ability to generate code for arbitrary architectures, dimensionality and integration methods.

By focusing on a specific type of simulation based, our STARGATES system is able to produce simulation code with serial and parallel computing performance that comparable to (very close to) that of hand-written code. The main advantage this generator provides is the ability to maintain a large number of equations and simulation methods without having to maintain a large number of code files. It also allows any number of simulations to be migrated to a new architecture with the addition of a single output generator. It is also useful in trying out different solver algorithm subtleties relatively quickly for a new PDE whereas hand generated code is generally much harder to debug and verify.

We have shown that this generator can produce code for three very different parallel architectures from the same simple equation description. These parallel implementations provide a significant speed-up over a single-core simulation showing that the code generated is efficient as well as correct.

We plan to extend STARGATES to cope with other potential parallel platforms such as OpenMP. There is also scope for adding further parallel-specific optimisations and heuristics to our code generator [32]. We also anticipate further developing the library of solvers and the support available for higher order calculus operations such as the bi and tri-harmonic operators. Finite-difference methods lend themselves well to problems that need high orders of accuracy and neighbour gathering stencil operations that extend beyond short neighbour ranges.

REFERENCES

[1] A. Mitchell and D. Griffiths, *The Finite Difference Method in Partial Differential Equations*. Wiley, 1980, no. ISBN 0-471-27641-3.

[2] A.I.Khan and B. Topping, "Parallel adaptive mesh generation," *Computing Systems in Engineering*, vol. 2, no. 1, pp. 75–101, 1991.

[3] D. Playne and K. Hawick, "Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs," in *Proc. CCGrid 2010, Melbourne, Australia*, no. CSTN-099, May 2010.

[4] R. F. Barrett, P. C. Roth, and S. W. Poole, "Finite difference stencils implemented using chapel," Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122, 2007.

[5] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)," *IEEE MIT-S Digest*, vol. WEIF-2, pp. 1033–1036, 2004.

[6] D. Greenspan and D. Schultzt, "Fast finite-difference solution of biharmonic problems," *Communications of the ACM*, vol. 15, no. 5, pp. 347–350, May 1972.

[7] C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., *Domain-Specific Program Generation*, ser. LNCS. Springer, 2003, no. 3016, ISBN 3-540-22119-0.

[8] P. McMullin, P. Milligan, and P. Corr, "Knowledge assisted code generation and analysis," in *High-Performance Computing and Networking*, ser. LNCS, vol. 1225, no. ISBN 978-3-540-62898-9. Springer, 1997, pp. 1030–1031.

[9] K. A. Hawick and D. P. Playne, "Automated and parallel code generation for finite-differencing stencils with arbitrary data types," in *Proc. Int. Conf. Computational Science, (ICCS), Workshop on Automated Program Generation for Computational Science, Amsterdam June 2010.*, no. CSTN-106, December 2010.

[10] S. K. Moore, "Multicore is bad news for supercomputers," *IEEE Spectrum*, vol. 45, no. 11, p. 11, 2008.

[11] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. April, p. 4, 1965.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994, ISBN 0-262-57104-8.

[13] W. Gropp, E. Lusk, N. Doss, and A. Sjkellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*. Argonne National Laboratories, 1996.

[14] *CUDA™ 3.1 Programming Guide*, NVIDIA® Corporation, 2010, last accessed August 2010. [Online]. Available: http://www.nvidia.com/

[15] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute Unified Device Architecture Application Suitability," *Computing in Science and Engineering*, vol. 11, pp. 16–26, 2009.

[16] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009, CSTN-065.

[17] J. Reinders, *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*, 1st ed. O'Reilly, 2007, no. ISBN 978-0596514808.

[18] Intel, *Intel(R) Threading Building Blocks Reference Manual*, 1st ed., Intel, July 2009.

[19] T. Parr, *The Definitive ANTLR Reference - Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007, no. ISBN 978-0-9787392-5-6.

[20] P. Milligan, R. McConnell, and T. Benson, "The Mathematician's Devil: An Experiment In Automating The Production Of Parallel Linear Algebra Software," in *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, Jan 1994, pp. 385–391, ISBN: 0-8186-5370-1.

[21] H. Bischof, S. Gorlatch, and R. Leshchinskiy, "Generic parallel programming using c++ templates and skeletons," in *Domain-Specific Program Generation*, 2003, pp. 107–126.

[22] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley, 2004, no. ISBN 0-201-88954-4.

[23] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[24] IEEE, *IEEE Std. 1003.1c-1995 thread extensions*, 1995.

[25] J. W. Cahn and J. E. Hilliard, "Free Energy of a Nonuniform System. I. Interfacial Free Energy," *The Journal of Chemical Physics*, vol. 28, no. 2, pp. 258–267, 1958.

[26] K. A. Hawick and D. P. Playne, "Modelling and visualizing the Cahn-Hilliard-Cook equation," in *Proceedings of 2008 International Conference on Modeling, Simulation and Visualization Methods (MSV'08)*, Las Vegas, Nevada, July 2008.

[27] D. Playne and K. Hawick, "Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA," in *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09) Las Vegas, USA.*, no. CSTN-073, 13-16 July 2009.

[28] V. L. Ginzburg and L. D. Landau, "(Published in English in Collected papers of L.D.Landau, Oxford Press, 1965, pp138-167)," *Zh. Eksp. Teor. Fiz.*, vol. 20, p. 1064, 1950, edited I.D. ter Haar.

[29] A. A. Abrikosov, "Ginzburg-Landau equations for the extended saddle-point model," *Phys. Rev. B*, vol. 56, no. 1, pp. 446–452, Jul 1997.

[30] K. A. Hawick and D. P. Playne, "Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA," Massey University, Tech. Rep. CSTN-070, January 2010, to appear in Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), 14-16 Feb. 2011, in Innsbruck, Austria.

[31] ——, "Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations," Computer Science, Massey University, Tech. Rep. CSTN-087, 2010, submitted to Springer J. Sci. Computing.

[32] C. Lengauer, "Program optimization in the domain of high-performance parallelism," in *Domain-Specific Program Generation*, 2003, pp. 73–91.