CMSE 822 Project Proposal

1 Introduction

Artificial life researchers design systems that exhibit properties of biological life in order to better understand their dynamics and, often, to apply these principles toward engineering applications such as artificial intelligence [Bedau, 2003]. Evolutionary transitions in individuality, which are key to the complexification and diversification of biological life [Smith and Szathmary, 1997], have been highlighted as key research targets with respect to the question of open-ended evolution [Ray, 1996, Banzhaf et al., 2016]. In an evolutionary transition of individuality, a new, more complex replicating entity is derived from the combination of cooperating replicating entities that have irrevocably entwined their long-term fates [West et al., 2015]. In particular, we focus on fraternal transition in individuality, events where closely-related kin come together or stay together to form a higher-level organism [Queller, 1997]. Eusocial insect colonies and multicellular organisms exemplify this phenomenon [Smith and Szathmary, 199

In previous work [Moreno and Ofria, 2018], we introduced the DISHTINY (DIStributed Hierarchical Transitions in IndividualitY) model, which seeks to achieve the evolution of transitions in individuality by explicitly registering organisms in cooperating groups that coordinate spatiotemporally to maximize the harvest of a resource. Detection of such a transition in DISHTINY is accomplished by identifying resource-sharing and reproductive division of labor among organisms registered to the same cooperating group. We designed this system such that hierarchal transitions across an arbitrary number of levels of individuality can be selected for and meaningfully detected. Furthermore, DISHTINY is decentralized and amenable to massive parallelization via distributed computing. We believe that such scalability is an essential consideration in the pursuit of artificial systems capable of generating complexity and novelty rivaling that of biological life via open-ended evolution [Ackley and Cannon, 2011, Ackley, 2016].

DISHTINY allows cell-like organisms to replicate across a toroidal grid. Over discrete timesteps ("updates"), the cells can collect a continuous-valued resource. Once sufficient resource has been accrued, cells may spend resource to place a daughter cell on an adjoining tile of the toroidal grid (i.e., reproduce), replacing any existing cell already there. As cells reproduce, they can choose to include offspring in the parent's cooperating "signaling channel" group or force offspring to create a new cooperating "signaling channel" group.

As shown at the top of Figure 1, resources appear at a single point then spread outwards update-by-update in a diamond-shaped wave, disappearing when the expanding wave reaches a predefined limit. Cells must be in a costly "activated" state to collect resource as it passes. The cell at the starting position of a resource wave is automatically activated, and will send the activate signal to neighboring cells on the same signaling channel. The newly activated cells, in turn, activate their own neighbors registered to the same signaling channel. Neighbors registered to other signaling channels do not activate. Each cell, after sending the activation signal, enters a temporary quiescent state so as not to reactivate from the signal. In this manner, cells sharing a signaling channel activate in concert with the expanding resource wave. As shown Figure $1a, b$, the rate of resource collection for a cell is determined by the size and shape of of its same-channel signaling network; small or fragmented same-channel signaling networks will frequently miss out on resource as it passes by.

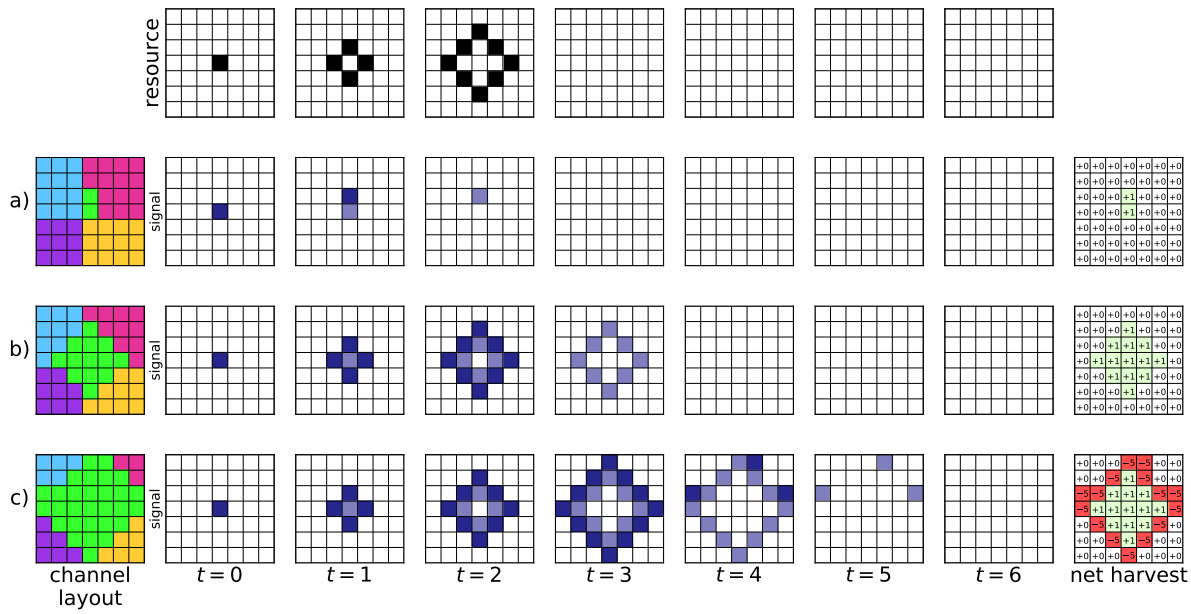


Figure 1: **Activation signaling, and net resource collection for three different-sized same-channel networks during a resource wave event.** At the top, a resource wave is depicted propagating over three updates and then ceasing for four updates (left to right). In row a, a small two-cell channel-signaling group (far left, in green) is activated; tracking the resource wave (top) yields a small net resource harvest (far right). In row b, an intermediate-sized 13-cell channel-signaling group yields a high net resource harvest. Finally, in row c, a large 29-cell channel-signaling group incurs a net negative resource harvest. In rows a, b, and c, dark purple indicates the active state, light purple indicates the quiescent state, and white indicates the ready state.

Each cell pays a resource cost when it activates. This cost is outweighed by the resource collected such that cells that activate in concert with a resource wave derive a net benefit. Recall, though, that resource waves have a limited extent. Cells that activate outside the extent of a resource wave or activate out of sync with the resource wave (due to an indirect path from the cell that originated the signal) pay the activation cost but collect no resource. Cells that frequently activate erroneously use up their resource and die. In our implementation, organisms that accrue a resource debt of -11 or greater are killed. This erroneous activation scenario is depicted in Figure 1c.

In this manner, "Goldilocks" — not to small and not too big — signaling networks are selected for. Based on a randomly chosen starting location, resource wave starting points (seeds) are tiled over the toroidal grid such that the extents of the resource waves touch, but do not overlap. All waves start and proceed synchronously; when they complete, the next resource waves are seeded. This process ensures that selection for "Goldilocks" same-channel signaling networks is uniformly distributed over the toroidal grid.

Cells control the size and shape of their same-channel signaling group through strategic reproduction. Three choices are afforded: whether to reproduce at all, where among the four adjoining tiles of the toroidal grid to place their offspring, and whether the offspring should be registered to the parent's signaling channel or be given a random channel ID (in the range 1 to 2^{22}). No guarantees are made about the uniqueness of a newly-generated channel ID, but chance collisions are rare.

In previous work [Moreno and Ofria, 2018], we used simple organisms that evolve parameters for a set of manually-designed strategies, to demonstrate that DISHTINY selects for genotypes that exhibit high-level individuality. Specifically, we observed

1. reproductive division of labor among members of the same channel (i.e., individuals enveloped in a same-channel signaling network ceded reproduction to those at the periphery),

2. cooperation between members of the same channel (i.e., pooling of resource on same-channel signaling networks),

3. reproductive bottlenecking (i.e., groups of cells sharing a channel ID descend from a single originator of that channel ID), and

4. suppression of somatic mutation via apoptosis coincident with high-level individuality.

We believe scale is fundamental to pursuing open-ended evolution in Artificial life. The DISHTINY model trivially scales to select for an arbitrary number of hierarchical levels of individuality through multiple separate, but overlaid, instantiations of this resource wave/channel-signaling scheme. Importantly, the model is designed in a decentralized manner and should scale well as additional computing resources are provided. Parallel computing is widely exploited in evolutionary computing, where subpopulations are farmed out for periods of isolated evolution or single genotypes are farmed out for fitness evaluation [Lin et al., 1994, Real et al., 2017]. DISHTINY presents a more fundamental parallelization potential: principled parallelization of the evolving individual phenotype at arbitrary scale (i.e., a high-level individual as a large collection of individual cells on the toroidal grid). Such parallelization will be key to realizing evolving computational systems with scale — and, perhaps, complexity — approaching biological systems.

2 Objectives

The objective of this project is to compare the parallel performance of alternate frameworks for the DISHTINY model: MPI and Charm++.

Charm++ requires the user to abstract their computation across of a set of Chare objects, written much like C++ objects. Chare objects, written much like C++ objects, interact by calling each other's invocation methods. These calls return immediately and, under the hood, the arguments to the call are wrapped into a message and then delivered to the Chare object. These objects are adaptively scheduled across available computational resources by Charm++'s runtime system. Charm++ is developed and maintained by the Parallel Programming Laboratory at the University of Illinois [Kale and Krishnan, 1993].

MPI defines syntax and semantics for the building blocks of parallel programs [Forum, 1994]. MPI executes the same code across many processors, with instances communicating through message-passing or remote memory access. Unlike Charm++, explicit decisions about precisely how MPI ranks will run and interact must be made by the programmer before compile time. MPI is a well-established standard with several mature implementations.

Comparing Charm++ with MPI allows exploration of the performance and behavior of synchronous versus asynchronous approaches to parallelizing DISHTNINY. One one end of the spectrum, individual toroidal grid tiles might proceed in lockstep update by update, with each tile waiting for all others to complete the current update before proceeding to the next. We implement this synchronous approach with MPI. On the other end of the spectrum, individual toroidal grid tiles might be allowed to proceed asynchronously with resource waves with any cell-to-cell communication proceeding as quickly (and potentially unevenly) as runtime conditions allow. We implement this asynchronous approach with Charm++.

For tractability, this work uses a stripped-down version of the DISHTINY model, simulating only resource-wave events and resource accumulation. In future work, we are eager to realize actual evolution with cell-like organisms capable of arbitrary computation via genetic programming in order to pursue more open-ended evolutionary experiments [Lalejini and Ofria, 2018]. Our scaling study will point the way forward for future development of parallel DISHTINY and the implementations put together for the project will provide a jumping-off point for that development.

3 Methods

3.1 MPI Implementation

3.1.1 Work Division

The MPI implementation chunks the tile grid into a set of distinct sub-grids, each of which is assigned to a single MPI ranks. Ranks are assigned to grid chunks using the built-in grid topology API, which helps to assure efficient distribution of neighboring ranks across the hardware topology.

Inside the update loop, each rank

1. initializes asynchronous send and receive calls to exchange information on current peripheral channel ID and resource wave activation states with neighboring ranks (e.g., `MPI_Isend` and `MPI_Irecv`),
2. updates the resource wave states of the interior of its sub-grid,
3. waits on all asynchronous receives to complete,
4. updates wave states of the periphery of its sub-grid, and
5. waits on all asynchronous sends to complete.

This ordering of operations allows for latency hiding: communication occurs asynchronously during the time when each rank is updating the resource wave states for the interior of its sub-grid. The, ideally, when each rank needs input from its neighbors (i.e., to update its periphery) that information is already ready and waiting.

Grid chunks are exactly uniform in size. (For simplicity of implementation, only grid sizes and rank counts that divide up evenly are allowed.) In this bare-bones implementation, channel IDs do not change at runtime. Channel configurations loaded from file contained uniformly-sized same-channel groups over the entire grid. The uniformity of channel configuration across the grid assures that the amount of computation needed to update each sub-grid is approximately equal. Each rank handles one sub-grid, so work should be uniform across ranks (i.e., load-balanced). However, in future work where channel configurations could become uneven during run-time, load balancing might become an challenge. For example, a channel configuration where much larger resource waves are possible might require slightly more compute time to update than a channel configuration with small same-channel groups. So, in this case, some ranks with faster channel configurations might regularly idle while waiting for ranks with slower channel configurations to update.

3.1.2 Input/Output

The MPI implementation exploits parallel HDF5 for input and output. Each process reads in its channel configuration from a HDF5 file containing the global channel configuration. Then, at the end of a run, each process writes its own resource accumulation values out to it's particular region of a HDF5 datafile (e.g., using `H5Pset_fapl_mpio`). This input/output scheme eliminates a potential bottleneck among MPI ranks at output time. Instead of having to take turns writing out data one at-a-time — or storing potentially large amounts of data in the memory of a single rank via a reduction, which would then have to be written out serially by that rank — all ranks can simply dump their data to directly to disk at will.

3.1.3 Memory Usage

MPI's distributed memory model allows the total amount of available memory to scale with rank count (e.g., as more nodes are added, more memory becomes available). So long as sub-grid size is kept constant (e.g., more ranks are added as problem size increases), the memory requirements of each rank will not increase as problem size increases. Additionally, because each sub-grid only communicates with a fixed number of neighbors (instead of communicating with all sub-grids), the memory required for communication overhead should scale in lockstep with available distributed memory as problem size increases and more ranks are added.

For the largest grids we tested with (8192×8192), the channel configuration file was hundreds of megabytes large. Scaling even larger, simply reading channel configuration into memory or writing out resource accumulations for the entire grid might become intractable without parallel and distributed I/O.

3.2 Charm++ Implementation

3.2.1 Work Division

In our implementation, each Charm++ Chare object represents a single grid tile. Each Chare engages in a self-update loop by means of a method in which it begins a resource-seed event, propagates a resource-wave signal to its neighbors (e.g., invokes a resource wave handling method on neighbor Chares), and then invokes the self-update method (which simply stores a message with instructions to run the self-update function in the Chare's queue). Resource-wave signals were prioritized over self-update calls in order to prevent accumulation of unhandled resource wave signals.

Unfortunately, the Charm++ scheduler did not schedule Chare's very uniformly under these conditions (there were 100-1000x disparities between the number of self-updates accomplished by different Chares). To fix this issue, we had each Chare invoke the self-update method of a neighbor so that updates cycled through the collection of tile Chares in a round-robin fashion. (Note that this strategy isn't serial because at one time there is a number of active update signals circling through the round-robin loop equal to the number of Chares present). Reported profiling results used this update strategy.

Work division between available processors is achieved using Charm++'s runtime scheduler, which runs each Chares' message handling functions on available CPUs. Charm++'s scheduler's seems somewhat opaque and guarded from direct influence by the end-user. For example, there does not seem to be a way to declare interconnect topologies at compile time. Instead, Charm++ uses runtime communication metrics to dynamically reassign Chare objects to different CPUs (e.g., see `http://charm.cs.uiuc.edu/research/topology`). Charm++ also uses runtime of migration of Chare objects to perform dynamic load balancing. In essence, Charm++ moves Chare objects from overutilized CPUs to underutilized CPUs in order to assure that computational resources are exploited efficiently. A variety of load-balancing strategies are available (e.g., see `http://charm.cs.illinois.edu/manuals/html/charm++/7.html`), ranging from centralized greedy scheduling algorithms to decentralized neighbor-swapping algorithms. Runtime migration requires the end-user to implement serialization and de-serialization routines for Chare objects, which was not tackled in this implementation, so the reported results do not take advantage of Charm++'s dynamic topology mapping or load balancing.

In addition to enabling runtime migration, better Charm++ performance might be attained by increasing the ratio of computation to communication. Instead of assigning an individual grid tile to each Chare, we might assign a larger sub-grid to each Chare, as we did with the MPI implementation. The presence of significant (instead of negligible) computation within each Chare's self-update method might allow for better latency hiding.

3.2.2 Input/Output

Input of channel configurations and output of resource accumulations was achieved very naively. The main Chare (the Charm++ analog to the main function) reads in the channel configuration for the entire grid at startup. Immediately following creation of the tile Chares, the main Chare tells each tile Chare its channel ID using a method invocation on the tile Chares. At the end of runtime, each tile Chare reports its resource accumulation back to

the main Chare. Once the last Chare has reported its resource accumulation, the main Chare writes all resource accumulation values out to a CSV file. Unfortunately, this output scheme was incompatible with the round-robin update invocation scheme described above, so verification results are reported using the self-update invocation scheme instead.

The naive input/output methodology used in our Charm++ implementation does not impact performance profiling because input/output and initialization time was not measured. If we were going to run the Charm++ code in production, however, more sophisticated I/O routines would be in order. A common Charm++ idiom appears to be use of built-in, optimized reductions to bring data in to a single I/O Chare, which then handles writing data to file.

3.2.3 Memory Usage

Using Charm++ Chares to represent individual grid tiles might lead to excessive memory use when simulating a large grid on a small number of CPUs. As grid size increases, the memory overhead associated with Charm++'s Chare book-keeping and queued messages between Chares increases in lockstep.

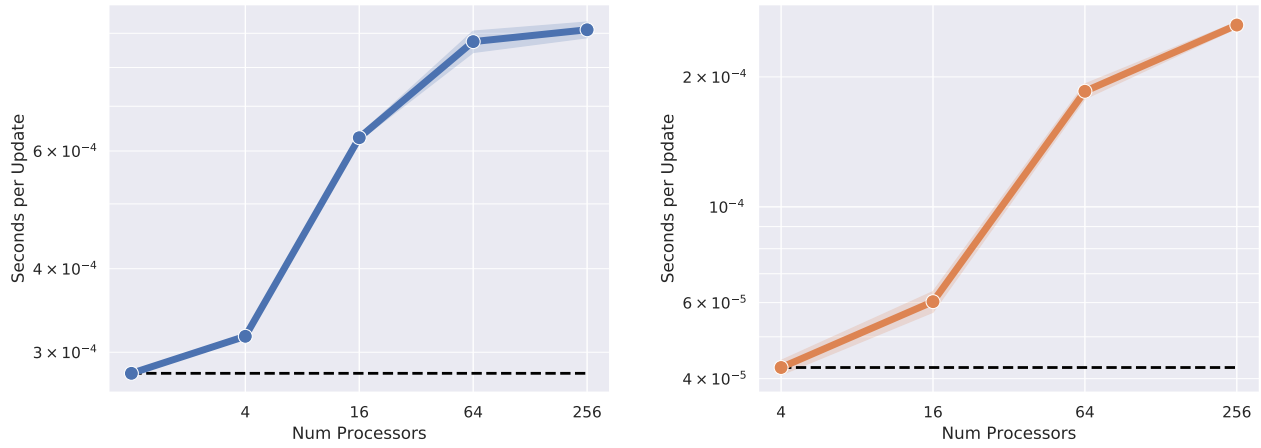
Charm++ employs a distributed memory model with individual Chares farmed out across nodes, so increasing the node count in tandem with increases in problem size should allow more grid tiles (and corresponding Chare objects) to be simulated without increasing the memory load on any individual node. However, the common Charm++ practice of piping I/O through a single Chare might prove problematic for large problem sizes if the amount of data that needs to be handled exceeds the memory available on a single node.

An additional memory issue that can arise with Charm++ is accumulation of un-processed messages. For example, if the self-update messages (which generate resource-wave signal messages and then generate another self-update message) are consistently prioritized in the queue over resource-wave signal messages, a growing backlog of resource-wave signal messages will bloat memory and eventually crash the run. Prioritizing resource-wave signal messages over self-update messages mitigates this issue.

3.3 Compute Environment

Performance experiments were performed on the Michigan State University High Performance Computing Center. Computing resources were requested using the following command: `salloc -n 256 --time=2:00:00`. This typically yielded a collection of processors spread across about 20 nodes (so, approximately 12 cores per node). Ideally, tests would have been performed using completely consistent compute configurations making use of entire nodes. To have enough resources allocated in a timely manner, however, experiments were performed using whatever resources the HPCC scheduler had available. However, for consistency's sake, all data part of a single analysis (e.g., presented on the same graph) were all generated using the same compute allocation.

Performance estimates were obtained by running our codes for a fixed time period (150 seconds) and then measuring elapsed updates. Initialization and input/output activities were not included in the 150 seconds of clocked runtime. The Charm++ elapsed update count was estimated as the mean of elapsed updates across all tile Chare objects.



(a) MPI implementation

(b) Charm++ implementation

Figure 2: Weak scaling analysis (time to solution versus parallelism with problem size proportional to parallelism) for MPI (a) and Charm++ (b) implementations. Dashed line indicates the ideal scaling relationship. Shaded area represents standard deviation of five replications for each observation. A problem size of 262,144 grid tiles per CPU was used for the MPI implementation and 1 grid tile per CPU was used for the Charm++ implementation.

Instructions to compile and run our code is provided in the `README.md` of the project code repository.

4 Results

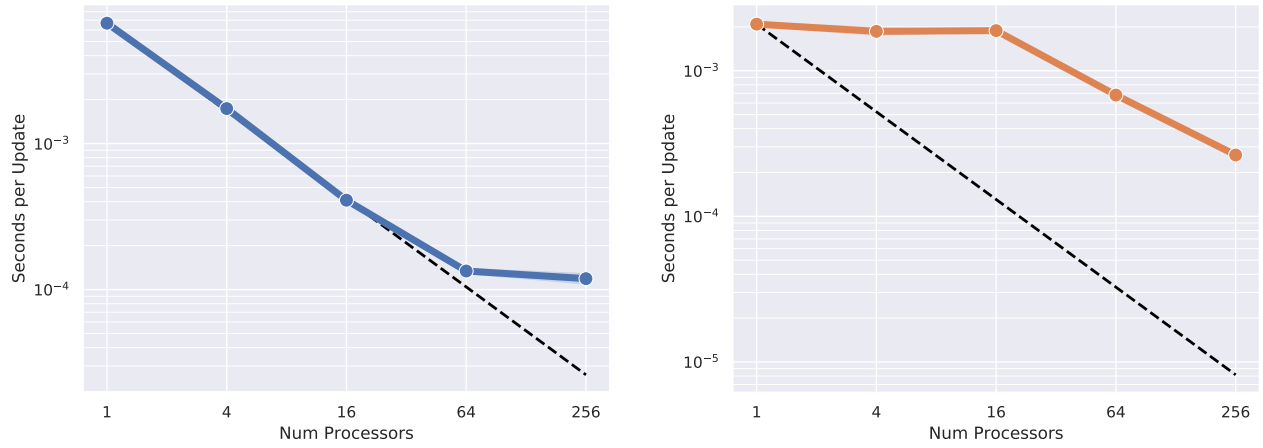
4.1 Weak Scaling

Figure 2 shows the relationship between time-to-solution and parallelism with problem size growing proportionally to parallelism for both the MPI and Charm++ implementations. Ideally, time-to-solution would remain unchanged as more work and more processors are added. However, both implementations experience a large performance hit scaling through the range of about 4 to 64 processors. As discussed in , the HPCC allocated a compute environment with about 12 processors per node. Thus, this large performance hit to both implementations is probably due to greater communication costs for communication between nodes. Encouragingly, though, both implementations (but especially MPI) see a less severe reduction in performance scaling from 64 to 256 nodes, suggesting that scaling up further to even more nodes could be accomplished efficiently.

4.2 Strong Scaling

Figure 3 shows the relationship between time-to-solution and parallelism with a fixed problem size for both the MPI and Charm++ implementations.

At an overall grid size of 2048×2048 MPI scales near-ideally up to 64 processors before



(a) MPI implementation

(b) Charm++ implementation

Figure 3: Strong scaling analysis (time to solution versus parallelism for a fixed problem size) for MPI (a) and Charm++ (b) implementations. Dashed line indicates the ideal scaling relationship. Shaded area represents standard deviation of five replications for each observation. Fixed problem size was a 2048×2048 grid for the MPI implementation and a 16×16 grid for the Charm++ implementation.

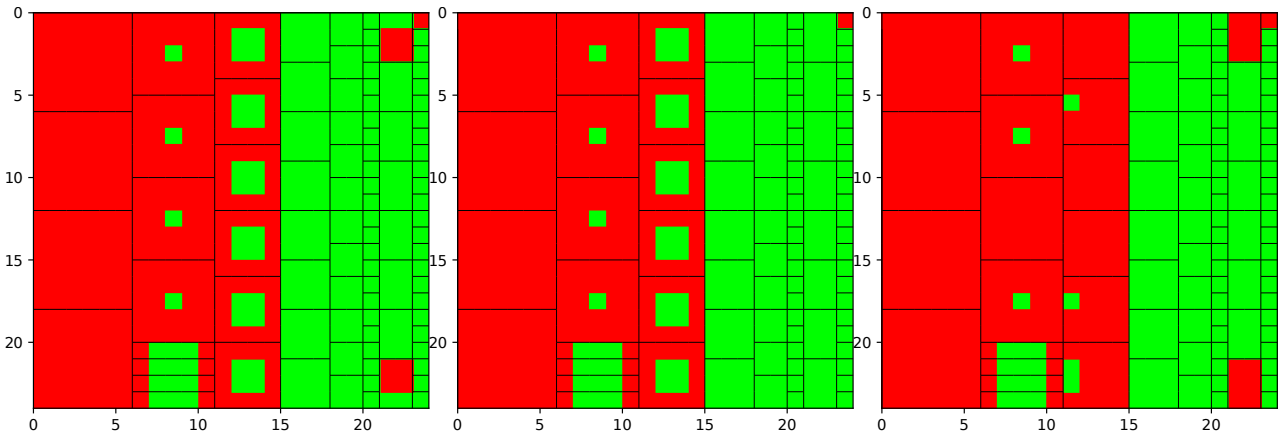
speedup levels off. At 256 processors, sub-grid size has decreased to 256×256. The performance saturation at 64 processors is perhaps due to a breakdown of latency hiding: at that smaller sub-grid size there isn't enough work for each rank to do while waiting for communications to complete.

Our Charm++ strong-scaling study employed a grid size of 16×16 (256 tiles total). Compared to the single-processor time-to-solution, Charm++ exhibits no increase in performance as the first 16 processors are added. Then, as more processors are added time-to-solution begins to decline at a near-ideal rate. At low processor counts, many Chare objects share the same CPU. The lack of speedup at low processor counts might be due to Charm++ overhead involved with to schedule and un-scheduling many Chares on the same CPU.

4.3 Verification

As with other endeavors in the field of Artificial Life [Bedau et al., 2000], our goal is to design and study an abstract system that experiences a phenomenon of interest rather than to directly model a specific (biological) system. Because our the aim is ultimately generative instead of strictly emulative, our criteria for correctness isn't necessarily as strictly defined as that for other scientific software.

Although ideally avoided for reproducibility's sake, our objective is not incompatible with nondeterminism introduced by race conditions. (In fact, demonstrating robustness to asynchrony and hardware hiccups/failure is perhaps useful for arguing that our approach is compatible with extreme computational scaling [Ackley, 2016].) At a fundamental level, our asynchronous Charm++ implementation is sensitive to hardware stochasticity (e.g., what order



(a) Charm++ (b) MPI $n = 1$ (c) MPI $n = 4$

Figure 4: Final resource accumulations as verification of implementation correctness for Charm++ implementation (a), MPI with $n = 1$ (b), and MPI with $n = 4$ (c). Black lines indicate divisions between same-channel groups. Red coloring indicates negative resource accumulation at a tile and green coloring indicates positive resource accumulation at a tile. Note that the single upper-right hand tile is part of a contiguous same-channel group with the large upper-left same-channel group via toroidal wraparound and that the top and bottom 3×2 same-channel groups in columns 22 and 23 are a contiguous same-channel group via toroidal wraparound.

Charm++ messages are received and queued in).

Here are our criteria for correctness:

1. medium-sized same-channel groups accumulate resource at a greater rate than small and too-large same-channel groups, and

2. for the MPI implementation, our simulation yields exactly the same results on any number of processors.

I verified the implementations by loading a manually-designed 24×24 channel layout and then inspecting resource accumulations at the end of a 150 second run. The channel layout was designed with a general left-to-right gradient of same-channel group size, with a few same-channel groups wrapped around the toroidal edges. Figure 4 shows resource accumulation results mapped over outlines of same-channel groups for the Charm++ implementation, for the MPI implementation with $n = 1$, and for the MPI implementation with $n = 4$. See the figure caption for details on same-channel groups wrapped around the toroidal edges.

For these runs, a resource wave size of three units was used. That means that any tile that is more than three grid steps away from another tile in its same-channel signaling group should experience negative resource accumulation. The Charm++ implementation appears entirely correct: much-too-large cell groups exhibit completely negative resource accumulation, slightly-too-large cell groups exhibit negative resource accumulation at the periphery, and medium/small cell groups exhibit positive resource accumulation. The MPI implementations are not entirely correct. In the $n = 1$ case, we don't have negative resource accumulation in the

too-large top-to-bottom-wrapped same-channel group. In the $n = 4$ case, we have several sites at the interior of slightly-too-large cell groups where negative resource accumulation occurs where we would have expected positive resource accumulation and vice-versa.

The MPI implementation needs more debugging for correctness before it would be ready for research use. However, the implementation suffices for meaningful performance profiling, which is the primary objective of the project at this stage.

5 Conclusion

The goal of this project is to compare asynchronous and synchronous approaches to parallelizing digital evolution experiments on fraternal transitions of individuality. The asynchronous approach was implemented with with Charm++ and the synchronous approach was implemented with MPI.

Charm++ provides an appealing message-driven paradigm would align nicely with existing software in our lab group designed for event-driven evolutionary computation [Lalejini and Ofria, 2018]. Exploiting Charm++'s object-oriented framework and simple method invocation to message syntax would allow for intuitive and rapid re-use of this existing software.

On the grounds of performance, though, MPI proves the better choice. The strong scaling of our MPI implementation is much closer to ideal than that of Charm++ implementation, particularly as the first 64 CPUs were added. In addition, although MPI implementation took a performance hit moving from single-node to multi-node execution, the near-identical times to solution when run on 64 and 256 cores in our weak scaling study suggests that we might be able to perform further scaling past 256 processors efficiently.

In addition, MPI allows for more explicit control of important aspects of parallel programming. For example, MPI allows the programmer to specify that ranks interact in a grid topology so that the placement. No such control is possible in Charm++; the programmer must instead rely upon the adaptive runtime system to pick up on communication patterns and migrate Chares appropriately Working with MPI will allow us to much more tightly fine-tune our software. It's faster and allows better control.

Finally, because MPI is much more ubiquitous than Charm++ it benefits from better support and integration with other software, such as parallel I/O with HDF5 or checkpoint-restart with DMCTP.

Although Charm++ offers an appealing object-oriented and event-driven software development framework, MPI is the obvious choice moving forward. We hope that exploiting parallel computing power will allow us to perform much larger-scale evolutionary transitions of individuality experiments in order to better understand these important evolutionary events.

6 Acknowledgements

This research was supported in part by NSF grants DEB-1655715 and DBI-0939454, and by Michigan State University through the computational resources provided by the Institute for Cyber-Enabled Research. This material is based upon work supported by the National Science

Foundation Graduate Research Fellowship under Grant No. DGE-1424871. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[Ackley, 2016] Ackley, D. H. (2016). Indefinite scalability for living computation. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*.

[Ackley and Cannon, 2011] Ackley, D. H. and Cannon, D. C. (2011). Pursue robust indefinite scalability. In *HotOS*.

[Banzhaf et al., 2016] Banzhaf, W., Baumgaertner, B., Beslon, G., Doursat, R., Foster, J. A., McMullin, B., De Melo, V. V., Miconi, T., Spector, L., Stepney, S., et al. (2016). Defining and simulating open-ended novelty: requirements, guidelines, and challenges. *Theory in Biosciences*, 135(3):131–161.

[Bedau, 2003] Bedau, M. A. (2003). Artificial life: organization, adaptation and complexity from the bottom up. *Trends in cognitive sciences*, 7(11):505–512.

[Bedau et al., 2000] Bedau, M. A., McCaskill, J. S., Packard, N. H., Rasmussen, S., Adami, C., Green, D. G., Ikegami, T., Kaneko, K., and Ray, T. S. (2000). Open problems in artificial life. *Artificial life*, 6(4):363–376.

[Forum, 1994] Forum, M. P. (1994). Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA.

[Kale and Krishnan, 1993] Kale, L. V. and Krishnan, S. (1993). Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM.

[Lalejini and Ofria, 2018] Lalejini, A. and Ofria, C. (2018). Evolving event-driven programs with signalgp. *arXiv preprint arXiv:1804.05445*.

[Lin et al., 1994] Lin, S.-C., Punch, W. F., and Goodman, E. D. (1994). Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 28–37. IEEE.

[Moreno and Ofria, 2018] Moreno, M. A. and Ofria, C. (2018). Toward open-ended fraternal transitions in individuality. *PeerJ Preprints*, 6:e27275v1.

[Queller, 1997] Queller, D. C. (1997). Cooperators since life began. *The Quarterly Review of Biology*, 72(2):184–188.

[Ray, 1996] Ray, T. S. (1996). Evolving parallel computation. *Complex Systems*, 10:229–237.

[Real et al., 2017] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911, International Convention Centre, Sydney, Australia. PMLR.

[Smith and Szathmary, 1997] Smith, J. and Szathmary, E. (1997). *The Major Transitions in Evolution*. OUP Oxford.

[West et al., 2015] West, S. A., Fisher, R. M., Gardner, A., and Kiers, E. T. (2015). Major evolutionary transitions in individuality. *Proceedings of the National Academy of Sciences*, 112(33):10112–10119.