Accelerating Agent-Based Ecosystem Models using the Cell Broadband Engine

Michael Lange
⟨michael.lange@imperial.ac.uk⟩
Department of Computing
Imperial College London

Supervisor: Dr. Tony Field ⟨ajf@doc.ic.ac.uk⟩
Second Marker: Prof. Paul Kelly ⟨p.kelly@doc.ic.ac.uk⟩

June 25, 2009

Abstract

The objective of this project is to investigate how dedicated Streaming architectures, such as the Cell Broadband Engine, can be used to speed up a class of *agent-based* models generated from a domain-specific problem-solving environment called the Virtual Ecology Workbench (VEW). This is a platform used by biological oceanographers to build models of ocean plankton ecosystem. The report presents a detailed performance analysis of existing *agent-based* VEW simulations and explores their mapping to parallel Stream Processing as offered by the Cell. We demonstrate the use of streamed multi-core arithmetic processing and show how it can be used to achieve a considerable performance increases in the agent update loop and substantial overall performance gains in models with modest numbers of agents. Based on this we develop a prototype code generator for the VEW specifically targeted at the Cell. We then give a full account of performance limitations imposed by sequential components of the VEW algorithm. In particular, we show that the VEW's existing *Particle Management* algorithm (which controls the number of agents in the simulation) is a major bottleneck to the parallelization of large-scale models. We end by discussing potential solutions to the *Particle Management* problem.

Acknowledgements

I would like to thank my supervisor Dr. Tony Field for all the interesting discussions and his invaluable support throughout this project. I would also like to give special thanks to Wes Hinsley for walking me through the VEW and helping me understand the algorithm. In addition I would like to extend my thanks to Wilhelm Kleiminger and Phil Meyer for providing the Toymodel code my project was based on. Furthermore I would like to mention Matteo Sinercia for creating the original LERM plankton model and John Woods who supervises the Virtual Ecology research group. Special thanks also go to my girlfriend Stefanie for supporting me throughout my studies.

Contents

Introduction

IBM recently set a new record in supercomputing power by breaking the Peta-FLOPS barrier (Floating Point Operations per Second) with the Roadrunner model. At the heart of this machine IBM used 12,000 Cell Broadband Engine (Cell BE) multi-core chips as ”accelerators” for arithmetic computation, on top of 7,000 conventional supercomputing cores. The Cell chips are added to improve the systems capability to crunch vast amounts of unstructured mesh data for scientific calculations. The parallel multi-core Streaming architecture of the Cell not only provides high FLOP rates, but also yields an impressive data throughput that is highly desired for applications handling large data volumes.

The Cell BE is easily accessible for example via the Sony Playstation 3 gaming console. Not only is this a very inexpensive way of getting a Cell processor, but Sony also has built in the opportunity to run different distributions of Linux on the Playstation 3. This enables developers to generate and run their own code for Cell processing on a PS3.

The idea of this project is to explore the application of the Cell to a class of agent-based simulations as generated by the Virtual Ecology Workbench (VEW). This is a scientific tool allowing biological oceanographers to build models of the plankton ecosystems within a simulated water column in the ocean. The VEW currently generates Java code to simulate the specified ecosystem from a model description.

These models have a number of potentially important scientific applications. They can give insights into the complex cause-effect relationships within a modeled ecosystem, allowing researchers gain a deeper understanding of the dynamics of the marine ecosystem. VEW models have many applications, for example understanding biodiversity of the ocean, the effect of marine plankton on the atmosphere and global climate, and the plankton ecosystem’s response to external influences, like fishing or pollution.

VEW simulations are based on Lagrangian Ensemble modeling [3] which models potentially very large numbers of individual agents, as compared to traditional population-based approaches that are based on the solution of relatively small sets of coupled differential equations. The agents act independently within the simulation. This requires individual updates of each agent during each timestep of the simulation, resulting in large amounts of arithmetic computation. We aim to exploit the inherent data-parallelism of agent-based LE simulations by utilizing parallel multi-core Stream Processing, as provided by the Cell BE architecture. Combined with the Cell’s superior arithmetic capabilities we hope to achieve significant performance improvements.

There are two phases to this project. The first phase is an investigation into the internal architecture of VEW models with the intent of finding an efficient mapping to the Cell BE that will improve the performance of VEW-generated simulations. This is a largely manual task and is based on the analysis of a reference model ("Toymodel") that is representative of those created by the VEW. The second phase is an investigation into the performance limitations of the current VEW algorithm imposed by sequential processing components.

The key incentive of this project is to create an efficient prototype simulation with a generic structure suitable for automated code generation from an abstract model description. This is a key challenge towards the inclusion of performance-optimized simulations into the VE Workbench.

The main contribution of this project is a partial model compiler for the Cell BE platform. This compiler generates efficient vector processing code for updating LE agents. During the development phase of the compiler several structural changes have been applied to the prototype simulation in order enable automated code-generation. This resulted in additional performance increase of the generated update code. The main contributions made towards the prototype simulation are detailed in chapter 4 and include:

- Full vectorization of update code
- Flexible allocation and efficient localization of agent blocks
- Agent homogeneity through multi-array memory organization

After applying the listed optimizations the Toymodel's performance suffers from limitations due to sequential components of the simulation loop. During the second phase of the project, we therefore investigate the two most significant sequential parts of the simulation and analyze their impact on overall performance. In chapter 5 we show how particular hardware features of the Cell BE limit performance scalability of the current implementation of the Particle Management process for very large simulations.

In section 5.2.2 we also show an implementation strategy that limits the execution time of one sequential component to a minimum. For this we show how to parallelize a previously PPE-based linear agent search by using metadata generated on the SPEs during the update cycle. We will then conclude by discussing the applicability of this method to the remaining sequential overhead.

Background

2.1 VEW

The Virtual Ecology Workbench is a software tool designed to automate modeling of virtual plankton ecosystems. The created models are agent based simulations based on the Lagrangian Ensemble (LE) meta-model[3]. In contrast to traditional population-based modeling this approach is based on individually computing the biological and bio-chemical behaviour of individual plankters and inferring population properties from the simulated sub-populations. As the bio-diversity of species and the simulated time-frame are potentially unlimited, the simulations are computationally very expensive and demand considerable programming knowledge.

To aid this, the VEW aims to automate the generation of the simulation from the meta-language Planktonica [4], which provides oceanographers with an easy way of specifying the behaviour and biological properties of individual plankton species in terms of primitive biological equations, based on reproducible laboratory experiments. Thus researchers can create complex individual-based plankton simulations without the need for conventional programming [7].

The VEW consists of several components for creating and running LE simulations and analyzing the generated output data. All these components are written in Java for cross-platform compatibility. A user-friendly GUI lets users define a model of plankton species and their functional behaviour, as well as initial and boundary conditions of the simulation. From this a meta-definition of the model is created in XML and this is compiled to Java with a code generating compiler.

The compiler creates a set of Java classes, which when compiled perform a timestep-based single-threaded simulation of the created model. During execution properties of the ecosystem are logged for later analysis. The properties of individual agents and the overall plankton population are updated in timesteps usually representing 30 minutes. Simulations can be arbitrarily long, and may simulate several years.

2.1.1 Lagrangian Ensemble Modeling

Lagrangian Ensemble modeling simulates individual agents within a virtual water column (Mesocosm), which can be static or drift around in the ocean. Each agent represents a sub-population of a particular plankton species. The agent models the behaviour of one individual plankter with an associated sub-population size. From these sub-populations field properties, such as demography and biofeedback, can be inferred. Hence, LE models combine individual-based and field-modeling, in contrast to traditional population-based box-model approaches [3].

2.1.2 Water Column and Physics

The simulated water column (Virtual Mesocosm) has a fixed depth of 500 meters and is divided into layers of 1m depth and 1km ^{2} horizontal area. There is an open boundary at the base of the column, allowing plankton and chemicals to sink into the deep ocean [3]. The upper meter of the column is furthermore sub-divided into more fine-grained layers, in order to model accurately the absorption of solar irradiance (see figure 2.1). This enables simulating light-dependent biological processes, such as photosynthesis. Combined with the effect of wind and other environmental fluxes, these processes are responsible for turbulent water flow in the upper layers of the column. For this reason chemicals and plankton particles above a certain threshold layer, the so-called turbocline, get mixed very frequently and rapidly. Below the threshold laminar flow causes particles to gradually sink.

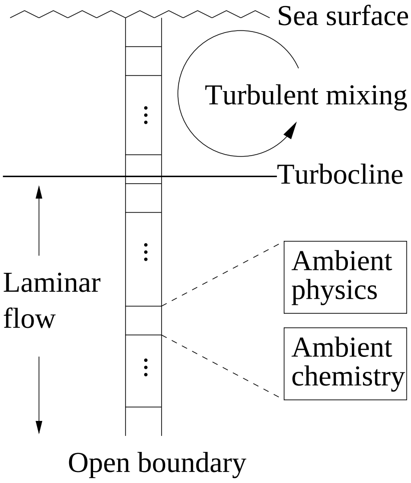


Figure 2.1: One-dimensional Water Column (taken from [9], with permission of authors).

The simulation time steps of 30 minutes are carefully chosen to, since they represent the approximate time needed turn over the upper mixing layers. The depth of these layers may vary seasonally during the course of the simulation, based on the flow fields generated by the column drifting in the ocean and the amount of light absorbed in the top layers of the column. Hence plankton populations in these layers may occlude this process and therefore affect the depth of the turbocline. This property of LE models is generally known as

Biofeedback, where simulated biological processes in the mesocosm affect the depth of the turbocline.

The vertical movement of the column in the ocean may also influence the flux fields responsible for changing the depth of the turbocline, and hence the track of the mesocosm is calculated iteratively with every timestep. Published meteorological data is then used to infer the effects and combined with Biofeedback used to calculate the depth of the mixing layers for the next time step. This process of determining the individual depths of the turbocline is computationally very costly overall. In order to speed up the calculations it is therefore possible to ignore Biofeedback effects, and read the physics data from previously executed simulations, representing stereotypical runs of a simulation for a given environment.

Within the column, since plankton does not possess the ability to actively move horizontally, the trajectories of individual plankters are only calculated in the vertical dimension and relative to the layers of the water column. These vertical trajectories are affected by turbulence which is modeled by random displacement of all plankters. This is the only Monte Carlo (random) aspect of the simulation. Different agents are said to follow different trajectories meaning that they experience independent random displacement over their lifetime.

2.1.3 Agents

In LE modeling each agent represents a sub-population of identical plankton particles based on the individual properties of one plankter. Each agent is part of a functional group which defines aspects of basic behaviour for sets of plankters in response to biological and local environmental properties[7]. The definitions consist of equations describing the fundamental bio-chemical and physical properties of the organism, derived from reproducible laboratory experiments. Thus each agent behaves as if it were one particle, but accounts for a sub-population of varying size.

Functional groups are sub-divided into particular species of plankters and biological stages, including potentially many living and one dead stage. An agents internal state consists of its stage, its associated sub-population and several internal state variables, usually representing internal pools of chemicals absorbed from the water. Together with environmental properties, such as depth within the water column, this defines an agent's state. A plankter's species and stage then define the individual agent's behaviour during each time step.

Since an agent's internal state is updated independently during each iteration, we can combine all these bio-chemical processes into one set of rules per species and stage, which in turn is applied to each agent of this type. Thus we are dealing with several heterogeneous sets of individual agents, each associated with one independent computational update function. Of particular interest now is that agents may non-deterministically change their internal biological state. Stream processing, on the other hand, requires agent homogeneity where it can apply one function to a large set of agents. Thus, it is vital to identify state changes immediately if agent homogeneity is to be preserved.

2.1.4 Particle Management

One of the advantages of LE models is the ability to limit the inherent error of the simulation by ensuring that a sufficient number of agents with individual trajectories exists for each plankton sub-population and stage in any given layer of the water column [3]. Agents whose population is less than a specified threshold are dropped from the simulation. In order to maintain a minimum number of agents of each type, the most populated agents of a sub-population are split in half and a new agent is created with independent trajectory. The independence of agents in general is ensured by basing their trajectory on random numbers.

Similarly, in order to limit the computational cost of an over-populated Virtual Ecosystem the Particle Management (PM) process may also merge the two least populated agents in a layer. The current run rules use a minimum and maximum threshold of agent sub-populations per layer to keep the agent population within reasonable bounds, without compromising the overall plankton demography of the ecosystem.

Particle Management is a sequential process at the moment that requires several search loops over parts of the agent set. For a simple Diatom model it is also the only function that actually creates and deletes agents and is therefore highly memory bound.

Normally for full models the PM runs after every iteration of the update loop. For the Diatom model we used during the investigation it is, however, possible to limit the process to only being run once a day (every 48 timesteps) without compromising the models accuracy.

Furthermore, a complete restructuring of the PM algorithm can have a strong impact on the bio-chemical stability of VEW models. We will therefore treat this component as unsuitable for parallel execution for the purpose of this report.

2.1.5 Agent Stage Management

A second sequential component gets added to the standard VEW loop as part of the vector-processing optimization required to achieve high data throughput on the Cell architecture. The arithmetic kernel function that is to be applied to LE agents is determined by an agents internal state, in particular by its biological stage. An agent, however, may non-deterministically change its internal stage during any given iteration.

On the other hand, streaming requires homogeneous sets of agents with identical kernel function for vector-based SIMD processing. Thus, agents that have changed their internal stage during the iteration need to be moved to an according memory area if we want to ensure the highest homogeneous throughput.

This process exhibits similar properties to the PM routines in that it is memory bound and sequential. In contrast to the PM, however, it has only one exhaustive linear lookup which can be exploited to parallelize the search.

2.1.6 Chemical Budgeting and Ingestion

LE agents continuously interchange chemicals with the surrounding environment through uptake() and release() function calls. The requested amounts are hereby stored as part of the agents state along with the internal pools of nutrient chemicals. This allows us in the next timestep to correct the amounts of nutrients interchanged by parameters calculated from the column field data.

Full VEW simulations also simulate predation between different plankton species. Although this is not featured in the prototype, the process of calculating Ingestion rates between agents in particular layers, as it featured in the Java version, follows similar principles.

The process is interleaved between multiple iterations by adapting multiple copies of agent variables with request parameters. The parameters get exchanged with the water column, which is simulated on the PPE after each iteration of the update loop.

In the model predator agents make requests for food in every layer they swim through during the iteration. These requests get averaged over the individual populations of the prey agent types for each layer the predator traversed. A parameter then scales the ingested nutrients down if insufficient food is available and adds the chemicals to the predator's pool in the next iteration.

VEW-based models also allow for Chemical Recycling and Budgeting, where specified amounts of chemicals are added to the virtual mesocosm at specified time-steps, or based on threshold values. This is done to counter the effect of chemicals collecting in particular layers and nutrients falling through the lower boundary of the column.

Ingestion and Chemical Recycling are not in the focus of this report, since they not part of the prototype simulation. Their computational structure is, however, similar to basic Nutrient Exchange. We will therefore assume that extending the model with these features will not have a substantial impact on the overall scalability and performance of the algorithm.

2.2 The Streaming Model of Parallel Computation

With the emergence of dedicated stream architectures, such as the Cell BE, the Stream Programming model is rapidly growing in popularity, as some of its main abstractions are natively implemented in the Cell hardware. The paradigm was originally developed for media and image processing tasks with simple, regular data access patterns and predictable control structures which performed repetitive arithmetic calculations over large volumes of data. More recently this approach has been mapped to general-purpose processors via frameworks, such as Streamware [11]. It has also been used to exploit the advantages of computational parallelism for scientific applications using irregular mesh constructs [13].

Stream Programming aims to decouple computation from memory access, in order to achieve task parallelism. According to [11] there are certain characteristics desirable for a Stream program, which include: large amounts of data to operate on, high arithmetic intensity, memory accesses that can be determined well in advance of their execution, and producer-consumer locality between computation functions. All of these features are found in VEW simulations, making them ideal candidates for parallelized Streaming Processing.

2.2.1 Streaming Architecture

Streaming Architecture contain several independent processing cores. In addition to a general-purpose CPU they consist of multiple data processing cores, called Processing Elements (PE), each of which maintains its own local memory area. These dedicated Streaming units are commonly optimized for fast arithmetic processing and high data throughput. SIMD (Single Instruction Multiple Data) or Vector Processing is often the key to achieving this high throughput. In the case of the Cell processor the PEs are not able to run an operating system. Thus the Master-Slave architecture of the Cell allows an OS to be run on the general-purpose CPU, while the PEs handle large volumes of arithmetic calculations.

Steaming architectures are commonly capable of asynchronous Direct Memory Access (DMA), where dedicated memory control units allow data transfer to happen in parallel to arithmetic processing. This allows for a constant data flow to and from the parallel processing units and requires a bus that provides efficient bandwidth for large volumes of data.

2.2.2 Data Localization and Parallelism

As the individual Processing Elements of Stream architectures cannot make arbitrary references to main memory, the data first has to be localized into the address space of the PEs. This process entails address renaming much like in conventional distributed systems. Within the Streaming paradigm data localization is based on a gather-compute-scatter approach, where data is arranged in continuous arrays of data structures (Streams) in main memory, which are bulk copied (gather) into the local memory area of each PE. The processing functions then work on these Strips of data before storing (scatter) the modified data array back into main memory.

The key to obtaining high parallelism during this process is to minimize data-transfer overheads through asynchronous DMA memory access that happens in parallel to computation. Thus, streamed data needs to be sufficiently buffered at the local processing unit in order to hide memory latencies. Double-Buffering is closely associated with Streaming Programming. This technique describes the use of two equivalent data buffers to mask memory latency, where processing is performed on one buffer, while data is pre-fetched into the second buffer. This process suffers one drawback, in that it does not mask data flow from the local elements to main memory. In section 4.2 we therefore use a third buffer in order to hide any data-transfer latencies in both directions.

Through asynchronous DMA Stream Programming attempts to turn memory latency into a bandwidth problem. It is therefore imperative for any Streaming architecture to provide high-bandwidth data transfer between it's parallel processors and shared memory. Thus, one of the Cell's main features is the high-performance bus (detailed in section 2.3.3) which provides high data throughput.

Stream Processing is targeted at utilizing data parallelism of a parallel algorithm for multi-core computation. Data parallelism is a result of identical operations being applied concurrently on different data items [2]. The amount of data parallel computations is very large in VEW-generated plankton models, due to the strong dominance of Agent Update computation in sequential VEW implementations. Since updates on a single agent's state are independent of other agents and repeated for large numbers of iterations, we can expect large parallel speedups when implementing a parallel Streaming algorithm.

2.2.3 Amdahl's Law

Amdahl's Law describes the effect sequential processing has on the potential speedups achievable by a parallel algorithm. It is thus one of the fundamental laws of parallel Computing.

If a problem of size W has a sequential component of size W_S, then W/W_S is an upper bound on its maximum speedup, no matter how many processing elements are used [2].

Thus, when analyzing the performance gains achievable on a parallel architecture like the Cell it is imperative to consider the overhead imposed on the overall performance by sequential processing. This is done in chapter 5 in this report.

2.3 Cell Broadband Engine

The architecture of the Cell cores is highly optimized for Stream Processing. It consists of 8 Synergistic Processing Elements (SPE) for SIMD Vector Processing connected to a central PowerPC Element (PPE) by the circular Element Interconnect Bus (EIB), as shown in Figure 2.2. The general-purpose PPE core allows for a PowerPC Linux distribution to be run on the Cell. For the purpose of this project we run Ubuntu 8.10 (PPC) Operating System on a Sony Playstation 3. The Playstation has 512MB integrated XDR memory, 256MB of which are available to a Linux kernel, as well as six of the eight SPE cores. There is no direct access for the OS to the on-board graphics card and the other two SPEs, one of which runs the Sony Hypervisor access restriction code.

As a Streaming architecture the Cell features a sophisticated data transfer system, in which the circular high-speed bus connects to all components on the chip. SPEs additionally have an associated 256KB Local Store for buffering the stream data. This local memory area has no cache hierarchy and holds code, as well data. The SPEs are equipped with a separate Memory Flow Controller (MFC) that coordinates asynchronous data exchange with main memory. Thus data transfer happens in parallel to arithmetic computation on the SPE.

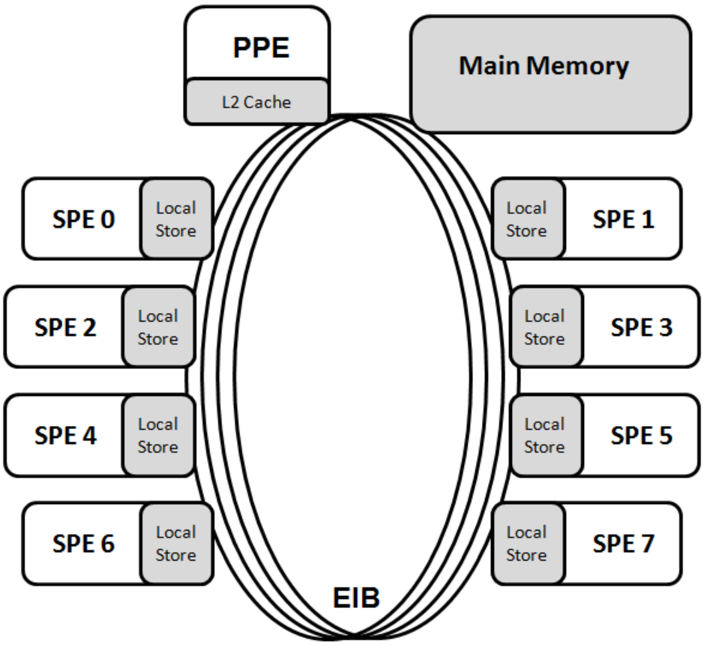


Figure 2.2: Conceptual Cell architecture, showing how the circular IEB connects all processing elements with main memory via four data rings.

The Cell is based on a Master-Slave architecture, where the PPE coordinates the overall application and delegates computationally intense processes to SPEs. Since the SPEs are highly optimized for vector-based floating-point-arithmetic they are slow when handling control structures. Therefore the PPE has to ensure that the data is layed out in an efficient format before passing it to the vector-processing units.

2.3.1 PPE

The Power Processing Element (PPE) acts as a system control processor that handles OS interrupts and manages data streams for external computation on the SPEs. It is the only processing core on the Cell chip that is capable of running an operating system. In our simulation it will delegate the arithmetically intense task of updating agents to the dedicated SPE arithmetic cores, whilst maintaining environment data for the virtual water column. Thus it is also responsible for all agent management tasks of the VEW algorithm.

The PPE is a dual-core general-purpose processor based on IBM's PowerPC architecture. As such it uses a typical CPU cache hierarchy including a 32KB L1 DCache and a 512KB L2 cache. Both caches use 128-bit cache lines and are connected to memory via the circular bus (EIB). The size of the cache lines is equivalent to the minimum amount of data transferable on the Cell's bus. One important property of the PPE's L2 cache is that it is *write-back*. This means that when the PPE makes a write request to an already cached location it will only update the L2 cache, but not write to memory. The cache will then write the data to main memory when the cache entry gets removed [1]. Through this the PPE attempts to reduce bus traffic by delaying all memory writes until they are necessary.

2.3.2 SPU

The Synergistic Processing Unit (SPU) is the functional core of the SPEs. It is programmed via intrinsic functions provided by libraries of the Cell SDK (Software Development Kit). These include memory flow controls for the MFC, message-passing primitives for mailboxes and vector instructions for arithmetic calculations and branching.

Arithmetic intrinsics come in two types: Atomic intrinsics, which get translated to single assembly instructions, and compound ones for more complex vector calculations. Compound intrinsics require specific coding to instruct the compiler to automatically in-line the according assembly instructions for performance optimization. All general arithmetic functions can be expressed using SIMD intrinsics, allowing for fully vectorized computation on four agents during the Agent Update phase of computation.

The purpose in optimizing the linearity of the generated SPU assembly is to utilize the dual-issue instruction pipelines of the SPU. The SPU pre-fetches instructions to in advance before issuing them on one of the two issue pipelines. The two instruction pipelines handle Load/Store and arithmetic instructions separately. One pipeline copies the relevant data fields into the 128-bit registers in the SPU's Register File and vice versa, while the second pipeline executes arithmetic operations between loaded registers. Thus it allows for two different instructions to be processed in one parallel processor tick.

The compiler will try to interleave the two types of instructions on sequential code parts in order to get a maximum dual-issue rate. Thus it is very important to write vectorized code in a sequential fashion to aid the compiler's attempts at optimization. Since the SPEs do not provide any type of processor caching further than the Local Store unit, we need to use predictive access schemes to improve data throughput from LS to the Register File.

The SPU also uses Branch-Prediction in their issue logic. That is that due to the instruction pre-fetching the code fetches only one branch of a conditional statement. If the execution branch is mispredicted, the SPU flushes its instruction buffer before continuing computation. This incurs a stall of 19 processor cycles during which no data is processed, whilst correct prediction has no overhead and results in continuous computation.

Branch prediction is used by the compiler to optimize loops, where only the loop exit results in a pipeline flush. Prediction intrinsics are used to tell the compiler how to predict the correct branch of execution. It is therefore very important to minimize the use of conditional within the SPU code and always predict the more likely outcome. The general solution to this problem is to compute both branches in full vectorized form and then select the individual vector items based on a vector of comparison flags.

A particular useful intrinsic tool provided by the SDK libraries is the SPU `decrementer()` function. This loads an integer number into a dedicated register, which decrements this number on every processor cycle. We can use this register to exactly measure the number of processor ticks gone since last setting the register. These can then be read within the code via intrinsics to get an accurate number of processor ticks since the `decrementer` was started. Hence we found a way of reliably timing individual sections of code. This is not only useful when generating optimized pipeline code, but can also provide us with execution times of individual Update functions, and measure stall times for channel communication.

SPEs communicate with other SPEs, the PPE and the memory units via so called channels. Synchronization is done via communication primitives which can be blocking or non blocking. For interacting with other SPEs and the PPE mailboxes are used. These are special-purpose registers and can encapsulate one integer in the message. The same type of channel is used for scheduling DMA to the MFC. SPUs schedule request for data transfers to the MFC via DMA intrinsics and then read a status register, usually blocking execution to ensure the data is available. Both types of channel communication may lead to channel stall, where the SPU spends its time idle. Thus if we have to consider both types of blocks when analyzing channel stalls.

2.3.3 EIB

The Element Interconnect Bus connects all components of the Cell core with each other and the memory in a circular fashion via four data rings. Each ring is 16 bytes wide and can carry several chunks of data, as long as they do not overlap physically on the ring. Data is transfered clockwise by two rings and anticlockwise by the other two. Therefore parallel reads and writes from memory are possible.

Each individual DMA operation takes 8 bus cycles and has a maximum of 128 bytes. A DMA transfer may consists of several operations up to an imposed limit of 16KB [1], which corresponds to 256 agents each containing 16`floats`. For data blocks larger than that several transfers have to be scheduled. Consequently the atomic unit of operation is 8 bus cycles long, meaning that only DMA transfers of less than 128 bytes utilize the rings non-optimally and are therefore inefficient. Thus we should not fetch less than 8 agents per DMA transfer.

The theoretical bandwidth of the bus is 204GB/sec, and can has been shown to be shown to transport up to 196GB/sec in artificial experiments with near perfect utilization [10]. This is however based on a perfectly circular DMA scheme where most of the data transfer happens between SPEs. At the current point of the project there is no evidence suggesting that execution performance is bound by data latency. Hence we have room to further utilize parallelism within the algorithm to transform data latency into a bandwidth problem through efficient localization.

Toymodel Analysis

The first step in our investigation into Cell-based VEW simulations is to analyze the performance characteristics of the existing Toymodel prototype. In order to establish possible areas of improvement, several profiling techniques have been used to identify bottlenecks, including the use of a Cell emulator developed by IBM (SystemSim).

This project was based on previous work done by a group of students investigating several approaches to parallelizing the VEW simulation code, including processing on Cell BE [5].

The Toymodel is based on a simplified Java version of the ”LERM-PS” simulation, which is based on the Lagrangian Ensemble Recruitment Model (LERM), developed in [6]. It simulates a statically anchored water column in the Azores region over the duration of two years. Figure 3.1 shows the biomass of living Phytoplankton within the simulated water column, sampled per week. It exhibits a VEW-characteristic curve, showing Diatom bloom during the summer and autumn period of each year.

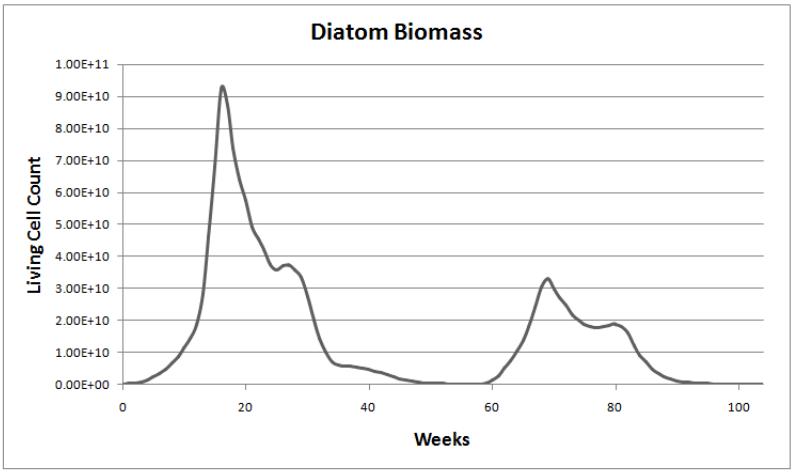


Figure 3.1: Toymodel result of 2 year Diatom simulation

The Toymodel represents a subset of the full LERM model. It only simulates Diatoms, which are plant-based plankters. Each Diatom can be in one of two stages, either *Living* or *Dead*. Zooplankton, like Copepods (animal-based plankters), as well as top-level predators are ignored by the model.

Predation between Copepods and Diatoms is not part of this simulation, so there is no need for the environment processes to model ingestion, ie. the process of one agent type feeding on another. Ingestion is revisited in section 7.3.1. Bio-optical feedback, the effect of plankton biomass on the Mixing Layer, is also not modeled. Instead the Physics parameters, such as ambient light and temperature, as well as other boundary conditions for chemicals are read from file. The file data is generated by the full Java LERM simulation.

However, the ability to add arbitrary plankton species, as well as other missing features are considered during the development of our prototype solution. In later sections we will show that all these features can be implemented with the existing approach without causing any significant performance decrease.

3.1 Execution Structure

The Toymodel simulation consists of two separate code-bases for PPE and SPE execution. The separation is needed since each platform has its own independent assembly format and compiler. The PPE acts as the main control thread by executing the simulation loop and handling general housekeeping tasks. The SPEs on the other hand act as dedicated Streaming units, only responsible for the agent update computation.

The PPE main loop executes all the necessary routines for simulating the environment processes within the water column prior to the agent update. It initiates the external agent update via channel communication with the SPEs. After the external update is completed, the PPE then performs the Particle Management process to keep the number of agents in the Virtual Ecosystem within specified boundaries. The structure of the PPE loop (Figure 3.2) is the same for all models generated by the VEW.

```
for (t = 0; t < 35040; t++) {
  readPhysics();
  mixChemistry();
  updateAgents();
  updateChemistry();
  particleManagement();
}
```

Figure 3.2: VEW main loop as executed on the PPE.

3.1.1 SPE

The pseudo code for SPE execution is depicted in Figure 3.3 and works as follows. Once the SPEs have received the signal to start the agent updates they first import a block of shared environment data from memory, which contains physics and chemistry data, specifically the depth of the Mixing Layer, the ambient water temperatures and the concentration of nutrient chemicals in each layer of the column. This data is constant throughout each iteration and only needs to be loaded at the beginning of the loop body (step 1 in Figure 3.4).

```
Receive starting signal
(1) Load shared environment data

while(work to do){
 (2) Pre-fetch agent block from memory
 (3) Process agent data
        if(4 agents of same type)
           (3.1) run vector update on 4 agents
        else
           (3.2) run scalar update on each agent
 (4) Store agent data
}
(5) Send environment data to PPE
Signal completion
```

Figure 3.3: Pseudo code for SPE execution.

Similarly, the SPEs record environment feedback data when traversing the agent array. The external units accumulate requests for nutrient chemicals for each column layer locally and synchronize these partial requests on the PPE after the iteration (see step 5 in Figure 3.4). The data accumulated across all six SPE units is aggregated at the PPE.

Agents are loaded into buffers and processed in blocks of fixed size. The allocation of blocks to SPUs is determined by hard-coded modular arithmetic within the SPE update loop and closely resembles Round-Robin. After the setup each SPE starts processing its associated subset of the agent stream, allocating two DMA buffers in the Local Store. The DMA loop follows basic Streaming principles in that it loads one block of agent data into a buffer, while processing the other pre-fetched block. Note that steps 2 and 3 in Figure 3.4 happen in parallel to each other. Using the `decrementer()` timing function we verified that only the first DMA block of each iteration incurs an idle wait (as expected), while each other DMA GET transfer is perfectly masked. This means that the data transfer than the vector computation and is completed before the MFC channel is checked, which then incurs less than 3 processor ticks.

In the reverse direction, all DMA PUT transfers, where the processed data is send back to memory, incur the full wait time for a bus cycle (eg. 200 ticks for 128 agents per buffer). The SPU blocks execution until each of those transfers is completed and processing is interrupted (step 4 in Figure 3.4).

When processing a block of agents (step 3 in Figure 3.4) the Toymodel utilizes the vector processing capabilities of the SPU. There are two update functions for each biological stage, one scalar function written in plain C (3.2), and one vectorized function using Cell vector intrinsics (3.1). The SPU inspects the stage of four consecutive agents in turn, and applies the vectorized function to them if they are all in the same stage. If the agents are in different stages each agent's update is computed individually by the scalar function. The same scalar function is also used to process the remaining agents of the last block that do not fit into a 4-agent vector.

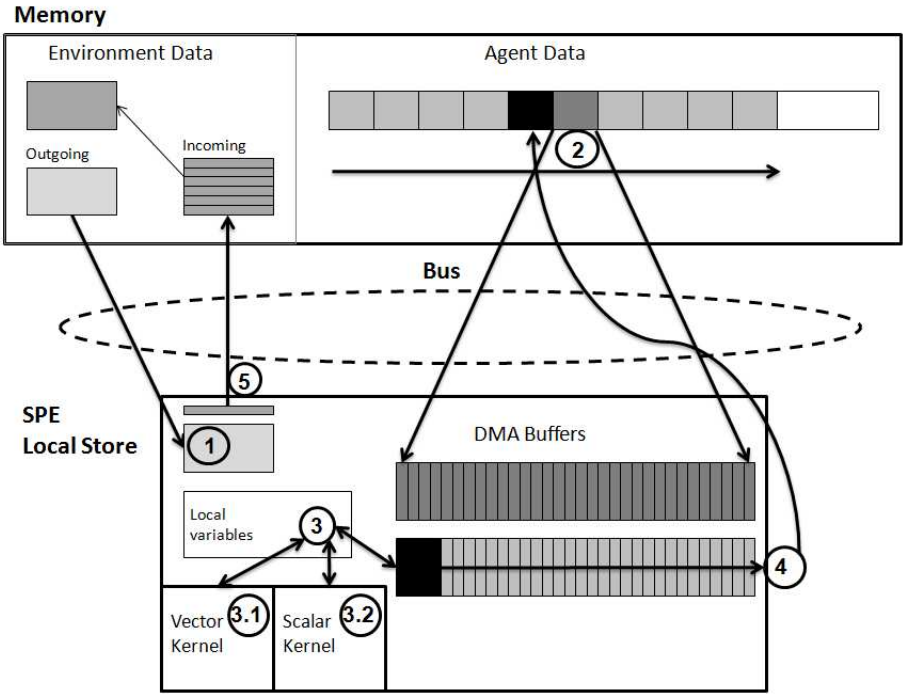
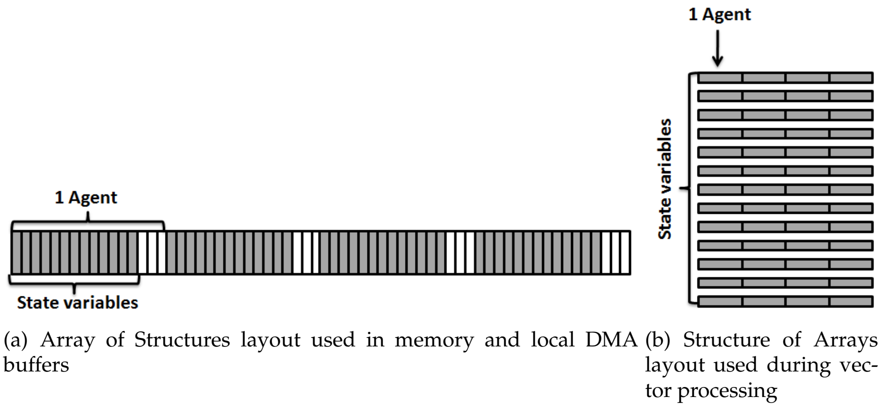


Figure 3.4: Conceptual data movement in the original prototype

3.1.2 Data Structure

In memory, agents are stored in a one-dimensional array. Even though an agent's state consists of 13 floating point numbers, a sub-array of 16 `floats` is allocated per agent. The alignment to 16 x n Bytes is required due to the 128 bit bus size. This corresponds conceptually to an Array of Structures (AOS) layout, where each agent represents an aligned structure.



(a) Array of Structures layout used in memory and local DMA buffers

(b) Structure of Arrays layout used during vector processing

Figure 3.5: Agent data layout in global and local memory.

The AOS structure is kept during the DMA transfer to the SPEs. SIMD processing on the SPU requires all data to be handled in vectors of four. When utilizing the vectorized update functions we therefore first need to copy the agent state variables to `vector float` types. Each vector now represents the same state variable for four agents, and the same transition is done on external environment variables. This data layout conceptually represents a Structure of Arrays (SOA), where the array length is fixed to four.

It is important to note that we always require all agents in a vector to be in the same biological stage, since this determines the update function we need to apply. In order to provide this agent homogeneity, an ordering loop was implemented on the PPE which places all dead Diatom agents at the end of the agent array in memory. This State Change loop was implemented as part of the Particle Management routine and involves redundant shuffling of dead Diatom agents in memory, resulting in a significant sequential processing overhead. Reducing this overhead is one of the main aims of the later parts of this project. The general homogeneity requirement for agent vectors implies a restructuring of the memory layout.

3.2 Performance

3.2.1 Scalability

In order to analyze the prototype's scalability we need to look at the execution time with increasing workload. For this we compared run times with the fastest sequential implementation of the Toymodel, which is written in C. Comparing the Cell to other processors is tricky, since its individual cores are clocked at different frequencies. We chose a 1.6GHz Intel Centrino dual-core processor with 2MB L2 cache and 2GB memory. 1.6GHz corresponds to the cited PPE clockspeed [10], which is the slowest of the Cell's processing cores. The Cell is usually quoted to run at 3.2GHz. This, however, corresponds to the clock speeds of the SPU's. A performance comparison against a reference CPU running at 3.2GHz is provided in section 6.2. We increased the workload on the x-axis by continuously doubling the number of initial agents for the same simulation, whilst scaling the Particle Management parameters accordingly.

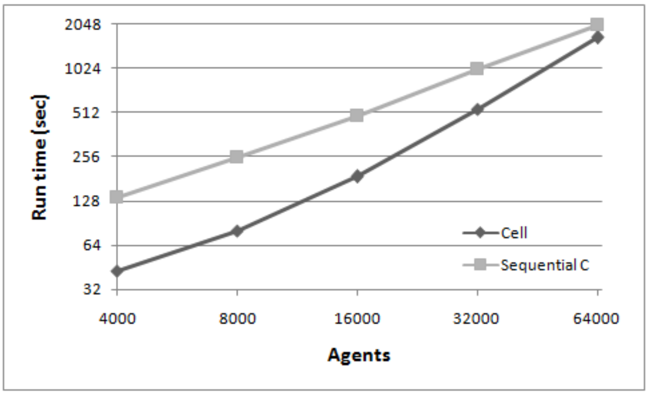


Figure 3.6: run times with increasing simulation size. Note: Both axis are exponentially labeled.

The results show that the current implementation does not scale well for large numbers of agents. Although it is comparably faster than the sequential implementation for the Toymodel's standard size of 4000 agents, the Cell algorithm shows a non-linear increase increase in execution time. The sequential version scales linearly, on the other hand. We can see that there is only a small performance difference for a 64000-agent simulation between the two versions.

3.2.2 SPU Timing

In order to evaluate the prototype exhaustively we also timed the individual update function (scalar and vectorized) and the stalls incurred by DMA operations. This was done using the SPU `decrementer()` tool, which gives micro-level timings in processor cycles (ticks). Due to the noise of potential dual-issue executions, `decrementer()` times may vary slightly. We therefore present average timing results.

Table 3.1 shows the dominance of vector processing when looking at the update times for Living Diatom agents. The time taken to perform one scalar agent update is far greater then the time required for a vector update processing four agents simultaneously. Due to many conditional statements in the scalar code the Living update produces greatly varying times, which is why we present a range instead of an average tick number.

When analyzing the DMA stalls in Table 3.2, we can see the effect of Double-Buffering for the data GET transfers. Compared to the increasing stall times of the PUT transfers with increasing buffer sizes, the GET transfers only stall for 2 ticks (the time taken to probe the MFC channel). We can therefore deduce that the buffering strategy successfully masks any data import from memory for the inner processing loop. However, the reverse is not true: storing the agent data back to memory leaves the SPU idle for a considerable amount of time.

Stage	Scalar		Vector	
	min	max	min	max
Living	504	786	210	218
Dead	168	175	120	124

Table 3.1: Execution times of vector and scalar Update functions in SPU cycles.

Agents	GET	PUT
32	2	20
64	2	34
128	2	66
256	2	140

Table 3.2: Waiting times due to DMA transfer latency for different buffer sizes, in SPU cycles.

3.2.3 SystemSim

In addition to Decrementer timings we also analyzed the performance of the Toymodel implementation with the IBM SystemSim simulator during the initial phase of the project. SystemSim runs a full emulation of the Cell BE core at flexible levels of detail. At the highest detail it can provide register contents of the SPE at run-time, including the execution pipeline of the SPU. Thus it can gather statistical information of the types of assembly instructions executed on the SPUs and give a detailed overview on execution stalls. The simulator can therefore give us detailed indications of the causes of the results presented above.

The simulation runs very slowly, since it is single-threaded and demands a lot of memory access. Due to long processing times we were not able to run any simulations past 8000 iterations, therefore giving only an incomplete picture of the simulation as a whole. However, due to the overall loop structure of the execution, the gathered data gives a good indication for further improvements of the code.

The data presented in Figure 3.2.3 was taken after simulating 8136 iterations of a 4000-agent simulation run on SystemSim. It represents the individual percentages of time spent executing instructions and stalling due to several causes. The causes are listed with their individual percentages of time stalled, and represent communication stalls on channels, stalls due to mispredicted branches and dependency stalls.

	Executed		Code Stalls		Communication
SPE	Single	Dual	Branch Miss	Dependency	Channel Stalls
2	25.3	6	9.9	22.9	31.6
3	24.2	5.6	9.8	19.9	36.2
4	26	6	10.5	21.6	31.4
5	27.5	6.3	11.1	23.1	27.2
6	28.2	6.5	11.4	23.5	25.4
7	28.9	6.6	11.7	24.2	23.6

Table 3.3: SystemSim results, showing percentages of execution time spent on individual instruction type

The presented data was obtained from running an improved version of the original implementation, which included triple-buffering, in order to hide all memory latencies (including PUT transfers). This is due to the fact that the simulator cannot differentiate between channel stalls due to data transfers and mailbox stalls created when waiting for PPE processing. For this simulation we manually verified that all inner-loop DMA transfers stall for less than four processor cycles using the `decrementer()` function.

Under this assumption the data presented in Figure 3.7 (based on Table 3.2.3) suggests that almost a third of the overall execution time is spent on the sequential parts of the simulation loop, while another third is spent on calculating update code. This identifies the Particle Management and associated agent array management as a performance bottleneck. The last third is then lost on different of stalls within the SPU instruction pipeline. Additionally, it should be noted that there are also a varying number of channel stalls, suggesting an uneven load distribution.

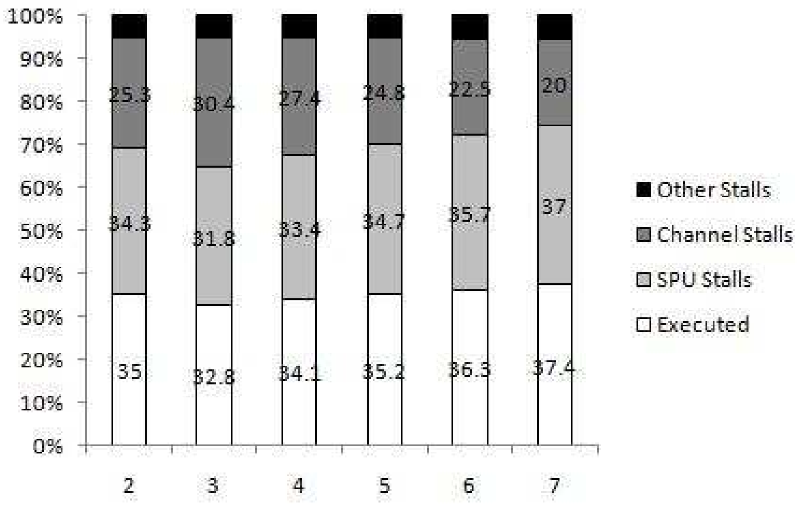


Figure 3.7: Percentages of the overall execution time taken up by executed instructions, code stalls and communication stalls on each SPU.

Within the instruction pipeline of the SPU branch and dependency stalls are created due to sub-optimal code preventing the compiler from optimizing register usage. Branch miss stalls are created when the whole pipeline has to be flushed due to a mispredicted if-statement. Dependency stalls on the other hand, refer to assembly instructions on one issue-pipeline (say the arithmetic pipeline) being stalled until the needed data is loaded into registers by the load/store pipeline. The SPU creates no-op cycles to fill these gaps until the other issue pipeline has caught up. The effect of this stalling process is quite significant, as the data suggests.

The simulator also differentiates between single instructions issued into one of the pipeline and dual-issues, where two instructions are processed concurrently by the SPU. The gathered data suggests a relatively weak utilization of this feature. This can be attributed to the remaining sequential parts of the update code, which quite possibly keep the compiler from optimizing more efficiently, and disrupt optimized vector processing. Attention should therefore be paid to this, when implementing a code generator for the vector update functions. We should aim to create SIMD vector instructions in a pure sequential structure, which would allow the compiler to optimize for instruction duality.

3.3 Evaluation

The SPU micro-timing analysis shows, as expected, that vectorized update code is significantly faster than scalar code. Our main goal therefore consists of re-structuring the prototype to only use vector updates. This also avoids the need to generate two types of update functions for a given agent stage.

The vector processing functions used in the implementation are not yet completely vectorized. There are two utility functions involved which calculate scalar parameters for each agents: the hard coded Random Number Generator and a conversion of physics parameters to the internal column layer structure. Additionally, a nested loop construct within the chemical pool update has not yet been vectorized. This accounts for the low dual-issue utilization found during the SystemSim analysis. We therefore need to manually vectorize the loop and utility functions before attempting an automated generation of this code.

There are two usages of the scalar update functions. For one we use scalar processing to update the remaining agents of the last block that do not fit into a vector of four. In order to overcome this scenario we will decouple the process of copying the agent data to the required `vector float` (SOA) format from the arithmetic processing. This is based on the fact that only the 13 state variables of an agent are stored back to memory. By decoupling the update *setup* and *teardown* routines we should thus be able to handle sets of less than four agents in vector format.

The second uses of the scalar updates is to process vectors of agents in different biological stages. Due to the Agent State Change handling on the PPE, which throughly orders the agents by stage, only one block during each iteration is heterogeneous. In order to eliminate this special case, we need to separate the two stages even further, so they will not overlap within a DMA block. This requires a major re-structuring of the agent layout in memory. We will therefore aim to adopt a 2-dimensional agent array layout, where all agents of one stage are kept in separate arrays. This will also ease the optimization of the Agent State Change handling itself, which currently is a performance bottleneck in the prototype.

This multi-array structure has far reaching implications for the Cell version of the code, since the current DMA localization loop is not able to handle several agent arrays. This is due to the hard coded allocation of blocks to SPEs within the DMA loop. This static allocation further prevents any experimentation with different scheduling strategies. These would be needed to overcome the load imbalance evident from the SystemSim analysis. A further argument for a multi-agent approach is that general VEW simulations usually involve several species with many more stages. Copepods alone, for example, have 16 different update functions. More complex VEW simulations would therefore profit from more flexible scheduling.

Agent Update Optimization

The development process described in this chapter is aimed at generating an efficient framework for SPE-based parallel Vector Processing of agent update code. The main target is the Model Compiler described in section 4.4. In order to achieve this, several structural changes have been applied to the existing Cell Toymodel. The applied changes additionally resulted in a significant performance improvement, particularly when scaled to simulations with large number of agents. The performance gains are detailed in section 4.5.

During development we largely focus on optimization of the SPE code. The aim is to avoid the use of wasteful scalar computation. We therefore explore SPU vector arithmetic and management, in order to fully vectorize the agent Updates. In addition to that, we demonstrate changes to the underlying data-transfer framework that enable a re-structuring of the agent layout in memory. The mentioned changes provides a more flexibly structured prototype which we can use as an execution harness for auto-generated vectorized agent Update code.

4.1 SPU Optimization

4.1.1 Vector Intrinsics

The vectorization of the original Toymodel Update code was partially complete. The old implementation used two utility functions, as well as a nested if-construct in scalar format, applied over all four agents of a vector. When processing scalars the SPU will use a full 128-bit register for any variable computed, resulting in wasted load/store instructions in the assembly code. For the compiler to optimize dual-issue execution we require a sequential code format handling each variable as a vector of four. This creates perfectly data-parallel processing and gives the compiler a chance to interleave load/store instructions with arithmetic.

Vector arithmetic is coded via SPU intrinsic functions provided by the Cell SDK libraries. Intrinsics can either be atomic, mapping to exactly one assembly instruction, or compound. The compiler will handle compound intrinsics as function calls, unless it is specifically told to in-line via a leading underscore '_'. Furthermore, for certain arithmetic intrinsics, like modulus-divide, two functions are defined, one for higher numerical accuracy and one for faster execution. The latter use the suffix `_fast`.

Nested If-statements can be treated like normal SPU conditionals, where the inner If is calculated in advance. As with any conditional on the SPEs, both branches are calculated and the results interleaved according to a vector of flags.

We can illustrate the performance gain attained from using these optimizations on the example of the random number generator function `rnd()` (Figures 4.1 and 4.2). The function is used to calculate the layer an agent moves to when it is above the turbocline. Applying the scalar function individually to all 4 agents in a vector takes 32 processor ticks. A call to the vectorized version not using in-lining or `_fast` implementations takes just 20 ticks. There are two modulus-divides and one `floor` rounding function used. Switching to `_fast` and in-lining these compound intrinsics reduces the execution time to 5 ticks. It is therefore sensible to include these low-level optimizations in any automated code generator.

```
static int seed = 100001;

static float rnd(float a) {
  seed += data.thread_id;
  seed = seed * 125;
  seed = seed % 2796203;

  float n = seed % (int)a;
  n += 1.0;
  return n;
}
```

Figure 4.1: Scalar RNG function

```
vector float seed =
  {100001.0f, 432543.0f, 780913.0f, 438901.0f};

vector float rnd(vector float _a){
  seed = spu_add(seed, spu_splats(data.thread_id));
  seed = spu_mul(seed, spu_splats(125.0f));
  seed = _fmodf4_fast(seed, spu_splats(2796203.0f));

  _a = _floorf4_fast( _a);
  vector float n = _fmodf4_fast(seed, _a);
  spu_add(n, spu_splats(1.0f));
  return n;
}
```

Figure 4.2: Vector RNG function

4.1.2 PRNG

The random-number-generator function represents the only Monte Carlo aspect of the simulation. It determines the displacement of Diatoms within the Mixing Layer due to turbulence. The use of a specified seed guarantees reproducibility of simulations on a sequential platform. On the parallel Cell

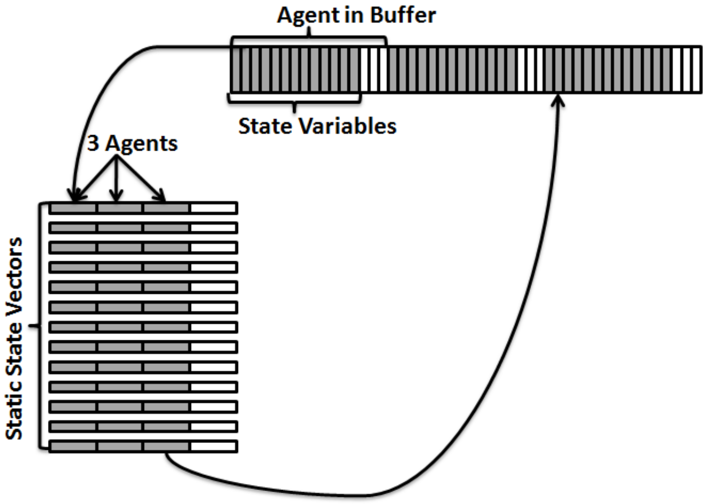


Figure 4.3: Padded vector conversion.

implementation, each local node (SPE) keeps its own counter. In order to guarantee different RNG sequences on all processors the internal SPE identifier (`thread_id`) is incorporated into the algorithm. As a side-effect of parallel computation, it is therefore not possible to re-create the exact results of a given sequential simulation.

During the later stages of the project it was discovered that the given vector implementation of the `rnd()` function did not provide a sufficient spread of random numbers. It was therefore decided to adopt the same algorithm the production VEW uses, a Mersenne Twister [8].

The Cell SDK provides an off-the-shelf library for Pseudo-Random-Number-Generation (PRNG) which contains implementations of two algorithms: Kirkpatrick-Stoll [1] and a Mersenne Twister. We ultimately chose to employ the MT generator, since it corresponds to the PRNG-method used in the full Java VEW. The Cell implementation works with low-level bit-shuffling [8] and are very efficient (the MT-generator function takes 6 SPU ticks).

4.1.3 Vector Conversion

One use of costly scalar functions arises from the need to handle sets of less than four agents. In order to use vector computation we instead ignore the remaining elements of a vector. This is similar to padding with zero values. It is vital for a code generator to separate agent and environment vectors from the local computation variables, in order to restrict the vector-to-buffer conversion. Knowing these, we can then create separate setup and teardown functions, parameterized with the number of agents computed. The skeleton functions populate and store the state vectors in a loop. This corresponds to traversing the vectors column-wise in Figure 4.3.

The full VEW Java simulator also uses generic setup and teardown functions for environment housekeeping tasks. These are defined in the XML model description via specific function calls, which the model compiler uses to identify environment variables. We can thus re-create the standard VEW layout by adding generic generic static placeholder vectors for each type of environment call and process the water column data within the same loop as the agent state data. This creates complete independence between the arithmetic parts of the update functions from the number of agents processed in a vector. The set of vectors shared between update and skeleton functions is defined statically in a separate header file.

```
for(m = 0; m < vectorSize; m++){
  /** Handling state variables **/
  if (__builtin_expect( (spu_extract(vars_STAGE, m)
      ==_STAGE_Living),TRUE)){
    var_data.livingDiatom += spu_extract(_c_new, m);
  }...

  /** Handling agent variables **/
  agentIndex = bufNo*AGENTS_PER_DMA+i + m;
  dma_buf[agentIndex][STAGE] = spu_extract(_stage, m);
  ...
};
```

Figure 4.4: Vector conversion of state and environment variables in loop.

In order to avoid branch-mismatches we also aim to use as few loops as possible. The housekeeping arithmetic accumulates particular agent fields, such as uptake requests for chemicals, or overall population. The accumulated fields correspond specified VEW functions, like nutrient uptake, and needs to be hard coded into the generation of the setup and teardown functions. The accumulation of environment properties is performed on a per-layer basis, and uses several conditional branches. At this stage it is vital to correctly predict the dominant branch taken for the conditional, in order to fine-tune simulation performance by avoiding SPU pipeline flushes.

Another low-level optimization was used in the original Toymodel code. If the vector length is known to be four, we can unroll the loop for vector population, allowing the SPU compiler to optimize data access. This yields small speedups, however is inconsistent with the padding solution. If fine tuning is needed the unrolled loop can always be added at a later stage, guarded by a conditional.

```
_stage = (vector float){
  dma_buf[bufNo*AGENTS_PER_DMA+pos][STAGE],
  dma_buf[bufNo*AGENTS_PER_DMA+pos+1][STAGE],
  dma_buf[bufNo*AGENTS_PER_DMA+pos+2][STAGE],
  dma_buf[bufNo*AGENTS_PER_DMA+pos+3][STAGE]
};
```

Figure 4.5: Conversion of agent state variable STAGE in unrolled fashion.

4.2 Data-Transfer Framework

4.2.1 Triple-Buffering

An easy performance gain was achieved by adding a third buffer to the traditional Double-Buffering DMA loop. This masks all DMA latencies due to PUT transfers. The same Triple-Buffering strategy was adopted for the second DMA loop, intended to transfer additional blocks of control data between SPE and PPE. For this we set up three buffers for each type of control data structure and use one common iteration counter to index the according buffers. We can thus synchronize the control buffers to the according agent data buffers.

4.2.2 Task and Feedback Farming

In order to provide more flexible load scheduling for the SPEs we decided to implement a centralized ”Farming” DMA framework. This aims to decouple the SPEs from scheduling and control by calculating the allocation of agent blocks onto processors on the PPE. This feature will not only allow us to schedule several agent arrays independently, but will also provide the possibility of investigating into more effective load-balancing schemes.

Since we aim for a centralized scheduling process we first need to find a way of transferring the relevant control data to the SPEs. This mainly includes memory address and size of the agent block to be processed. We therefore introduce a new Task Block (TB) data structure which describes one block of agents. A 2-dimensional array of Task Blocks is held in memory, one array for each SPE, allowing a scheduling process to allocate tasks by pre-calculating the relevant addresses and block sizes. Since the number of blocks to be processed changes dynamically with each iteration, we allocate the maximum array size possible and keep dynamic counters associated for each Task array. The same over-allocation method is used for allocating memory space to agents.

It is important to note that a block’s address needs to be known in advance to pre-fetching the data into the GET buffer. For this reason a 2-step-lookahead scheme was adopted (Figure 4.6). When processing agent block n, we pre-fetch data block $n + 1$, whilst fetching the address of block $n + 2$. Due to the asynchronous DMA provided by the Memory Flow Controller, we can schedule the address fetching in parallel to update processing and agent data transfers. This effectively masks all memory latency of fetching Task Blocks, apart from the two initial accesses.

Since TBs are relatively light-weight compared to agent data we can in theory add more control data to them. During the course of this project we have not found the bus to be a performance bottleneck. To verify this assumption checked that the added data transfer did not compromise any latency hiding.

Similarly to Task Blocks we introduced Feedback Block structures (FB). These contain data gathered by the SPEs that can be used for environment processing and agent management on the PPE. We utilize this feature in a later section for improving the Agent State Change loop. This follows the general Streaming principle of turning a computational problem into a bandwidth problem.

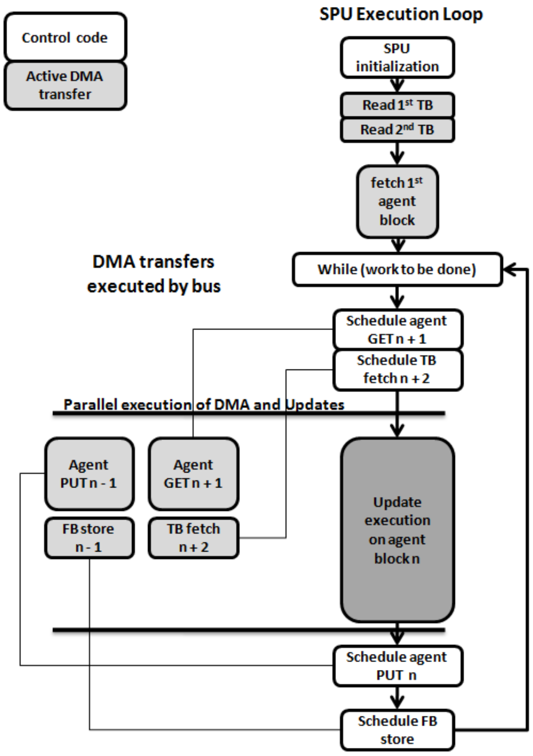


Figure 4.6: Conceptual data transfer loop executed on SPE with associated parallel DMA.

4.2.3 Scheduler

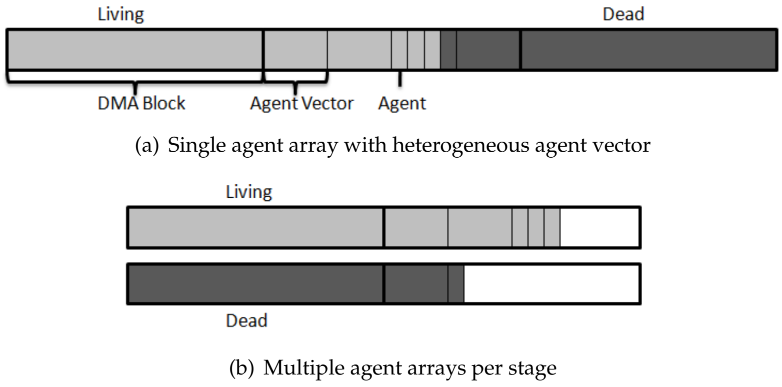
The `Scheduler()` process is run on the PPE before invoking the agent update loop on all SPEs. It traverses individual agent arrays, calculating memory addresses for agent blocks, and allocates them to the appropriate SPE task queue. It approximates a global Round-Robin scheme very similar to the original implementation.

The main advantage of the scheduler is that it can handle several arrays of agents. It keeps a shared count of the length of each task queue, allowing it to append tasks indefinitely. The current implementation marks the last block in each queue via a flag in the TBs, signaling the end of the update loop to the SPEs.

4.3 Memory Organization

Streaming architectures require buffers of homogeneous agents, in order to process each buffer generically with one associated update function. In order to exploit Streaming for optimized update computation, all parallel VEW implementations should therefore adapt a memory structure which separates agents by their internal state (biological stage). For this project we use a two-dimensional array of agents, with each sub-array holding agents of one particular stage.

With the availability of a flexible scheduling mechanism, coupled with a dynamic Farming DMA cycle, we are able to separate agents by their internal state. We can thus revise the memory structure, creating a set of homogeneous agent arrays. This eliminates the last usage of scalar update functions for heterogeneous agent vector (see Figure 4.7.a), and paves the way for automated update code generation.



(a) Single agent array with heterogeneous agent vector

(b) Multiple agent arrays per stage

Figure 4.7: Agent layout in memory

One key property of VEW simulations is that agents may change their biological stage dynamically during the run of a simulation. This state change of LE agents may occur non-deterministically from the point of view of the execution framework. For this reason the separation of agents by biological stage has to be revised after each iteration. In order to detect random stage changes the previous implementation used a sequential loop traversing the complete agent array, placing all dead Diatom agents at the end of the array. Since this loop ignored previous offset counts of dead agents it would inevitable also traverse the already dead section of the array. This resulted in a wasteful and memory-intensive shuffling of dead agents. The natural separation of the two agent types now prevented this, yielding additional performance increase. The immediate detection of agent state changes is a particular parallel artefact of multi-core VEW implementations. We shall show how to reduce this overhead even further in the next chapter.

Implementing the change in memory layout also has quite a significant impact on the sequential parts of the simulation, in particular Particle Management (PM). All PM search routines now need to be parameterized with the agent sub-array, resulting in smaller search spaces. The effect of this is unfortunately not very obvious in our Toymodel example, since it simulates a very large number of living Diatoms, compared to relatively few dead agents. In general, however, the stage populations may be arbitrarily balanced across different species and stages of the full VEW implementation. A more evenly balanced distribution of agents across arrays will have a positive impact on PM performance.

In order to discuss potential memory layouts for the VEW algorithm we have to keep in mind that the VEW stores agents by layer in the column because most environment processes, like nutrient exchange and predation ingestion, are calculated based on the layers an agent traverses during one timestep. This approach, however, is inapplicable to parallel streaming for several reasons. Firstly there are usually not enough agents in one layer to fill a buffer.

More importantly, though, agents constantly move, which would result in frequent re-allocation of agents within the multi-dimensional array. Due to the dependence of the update kernel, a stage separated layout is therefore the most appropriate for streamed update processing.

The layer-based environment data, however, is still needed for s sequential PPE processing. Due to dynamically changing agent states it is accumulated on the SPEs, since they perform an exhaustive traversal of all agents. Of particular interest is hereby the `agentsPerLayer` counter, which is used by the Particle Management process in conjunction with the overall agent count per stage-array. Since the array-length per stage is maintained on the PPE (SPEs do not create or remove agents), these two parameters need to be tightly synchronized. Otherwise the PM will remove incorrect agents from the simulation, which caused frequent problems during development. The effect of this happening was usually not visible in the output until several thousand iterations later in the simulation. For this reason, a lot of debugging effort went into this stage of the development process.

4.4 Model Compiler

After re-structuring the execution framework and eliminating scalar update functions, we were able to re-define the prototype's file structure. This allowed us to separate all Diatom specific update code and model parameterizations from the execution kernel on the SPE side. Subsequently, we were able to implement a generic code generator which creates the plankton specific vectorized update arithmetic code from an XML model description. This was done by extending the current Java Model Compiler.

The Model Compiler creates a fully vectorized encoding of the arithmetic computation specified in the model description file for plankton species and stage. It also creates generic `setup()` and `teardown()` functions specific to different species of plankton agents. The code generator creates agent-specific files that can be plugged into the prototype code base, resulting in a Cell-based VEW simulation similar to the Toymodel prototype. For the development process we used a Diatom-only version of the LERM [6] simulation the Toymodel is based on. After decoupling the Update arithmetic from SPU kernel execution our prototype simulation can be used as template code to generate the rest of the required simulation code.

Several performance issues still exist within the sequential PPE processing components. However, we chose not to pursue this route, in favour of a thorough investigation into the performance properties of the sequential PPE code components, in order to attempt further performance speedups (chapter 5).

4.4.1 VEW Model Compilation

VEW model descriptions are coded in XML by the VEW GUI and describe in detail all aspects of the simulation to be generated. The VEW model separates plankton agents by Functional Groups, which usually represent particular species of plankters. Each group has associated with it many update equations, each modeling a specific biological process, like respiration and photosynthesis. Every species further defines several internal stages an agent can be in, and maps individual biological equations to each stage. The Cell model compiler then accumulates the equations in order to create one vectorized update function. The SPU kernel then applies this function to the agent's internal state during each timestep of the simulation. An agent's state consists of several variables defined in the model description and is common to each agent of a particular Functional Group.

Since the set of internal state variables is specific to each Functional Group, the setup and teardown functions need to part of the model generation. The Model Compiler thus needs to maintain separate lists of environment, agent and local variables for each agent type. The environment and agent variables are defined statically in a global header file, accumulating all variables needed by all Functional Groups.

The modeling language Planktonica [4] is used to code the biological primitive equations which form the update code. In addition to standard mathematical expressions it also provides several environment functions. These are used by agents to handle interaction with the water column. Since requests are handled on the PPE between iterations, the request data is buffered per agent as part of an agent's internal state. Environment request data also gets accumulated per column layer on the SPEs and send back to the PPE for sequential processing at the beginning of the next iteration. Therefore, we need to accumulate the requests made during an update in the teardown function (see section 4.1.3). The request accumulation is a generic process for each environment variable, where the compiler adds generic housekeeping code for defined sets of variables. However, each provided environment function needs hard coding into the teardown generator function.

The full Java implementation of the VEW features several environment feedback loops. One loop, the uptake and release of nutrient chemicals into the water, is also included in the Toymodel prototype. For nutrient uptake, the simulation buffers the requested amounts of chemicals per agent in the agent's internal state and accumulates the requests for each layer. At the beginning of the next iteration, the requested amounts are evaluated on the PPE, and a parameter is published to all SPEs which describes the rate at which the requests have to be adjusted for each column, based on nutrient availability. The SPE code then adjusts the requested amount accordingly.

There are several further feedback loops with a very similar structure to this in the full VEW, most importantly ingestion rates due to Predation. However, since all these loops are calculated per layer and are adjusted between iterations, the general accumulation setup of the Nutrient uptake functionality can generically be extended to this.

4.5 Performance evaluation

As a result of Update code vectorization, the performance of the prototype simulation increased significantly. We will now analyze the execution times of the improved Toymodel code and it's scalability for increasing numbers of agents. In order to establish further areas of improvement we will also profile the individual run times of the update loop in comparison to sequential processing components run on the PPE. This will guide the investigation into performance gains from PPE execution in chapter 5.

During the analysis we compare the current Cell prototype to a sequential C implementation of the Toymodel simulation. This code is also a product of [5] and represents the fastest known sequential implementation of this particular simulation. Furthermore, the C implementation uses a similar agent array layout in memory, with separate arrays for agent of different stages.

Finding a reference processor for run time comparisons with the Cell is hard, since the different cores on the Cell chip run at different clock speeds. The SPEs are quoted to run at 3.2GHz, whereas the PPE is clocked at 1.6GHz. For our experiments we chose a 1.6GHz Intel Centrino dual-core CPU with a 2MB shared L2 cache and 2GB RAM. The 1.6Ghz clock speed was chosen in order to match the slowest Cell core (PPE). The choice of a dual-core reference architecture was motivated by the fact that the SPEs cannot run an operating system. The dual-core architecture generally runs the OS on a different thread to the updates, in the same manner as the PPE runs the OS separate from the SPEs. The OS chosen for both platforms is Ubuntu Linux version 8.10.

We will also compare the achieved results to the original Cell implementation of the Toymodel. For the purpose of scalability analysis the x and y-axis on all diagrams are labeled exponentially, in order to identify trends with increasing model sizes. We quote the initial number of agents on the x-axis. It should be noted that the agent size of the simulation varies over time, but follows the same trend for each simulation of the same type. We can therefore assume that the true simulation size is still linearly growing with the initial number of agents (see section 6.1.2).

The run times presented in Figure 4.8 suggest a very clear performance improvement over the original Cell implementation, as well as the sequential code. The structural optimizations applied not only yield the expected linear improvement, but also lowered the scalability gradient significantly. It is important to note, however, that the obtained execution times still form a curve, and hence the simulation does not scale linearly for large simulations. We can therefore suggest that for very large numbers of agents the sequential implementation will remain the most efficient, since it seems to scale perfectly linear. On the other hand, the importance of a separated agent structure in memory, as well as consistently vectorized updates, is highlighted by the performance gain achieved the original Cell code.

The optimizations applied to the SPU data-transfer loop (section 4.2), as well as the full vectorization of the update code (section 4.1.1) are responsible for linear performance increase. This corresponds to a downward translation along the y-axis in Figure 4.8. The same vertical translation can also be seen on Figure 4.9, which furthermore shows that the update code is a linearly scalable process.

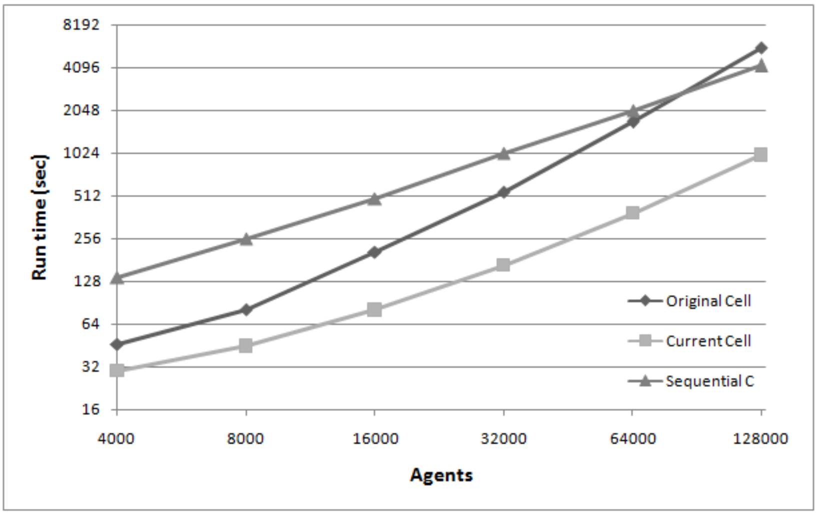


Figure 4.8: Execution times for Toymodel simulation on Cell and sequential C code.

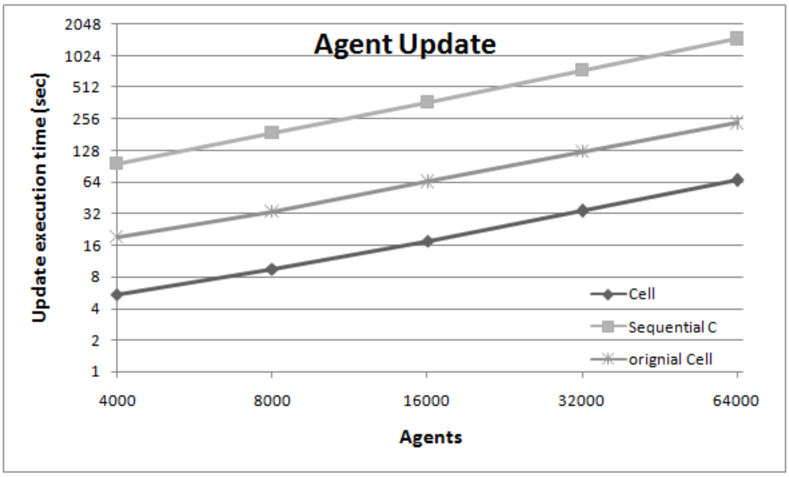


Figure 4.9: Update execution times of Cell and sequential C implementation.

In addition to the linear speedup, the separation of agent arrays described in section 4.3 lowers the gradient of the run time increase shown in Figure 4.8. This is best illustrated by comparing the individual execution times of the sequential Particle Management component of the algorithm. This process traverses the complete agent array several times, in order to identify particular agents to manage. The PM search routines are invoked per biological agent-stage though, resulting in unwanted large search spaces for a single-agent-array setup. In particular the Toymodel PM rules heavily merge *Dead* agents, in order to keep their numbers low. Thus the separation of agent-array types now causes the *Dead* agent search space to decrease drastically, since the number of *Living* agents is usually greater by several orders of magnitude. This reduces PM run time in a non-linear fashion and lowers the scalability gradient, as can be seen in Figure 4.10.

It is also important to note that, although strongly improved, the Particle Management routines are much slower on the Cell than on the x86 architecture (Figure 4.10). A close look also suggests that the sequential components are still a source of non-linear run time increase. This causes the overall efficiency of the parallel VEW algorithm to decrease constantly, resulting in non-linear performance scalability. This is the result of Amdahl's law (section 2.2.3), which dictates that the sequential parts of any algorithm limit the parallel scalability. In order to find the source of the non-linear performance decrease we therefore have to look closer at the sequential PPE processes of the algorithm.

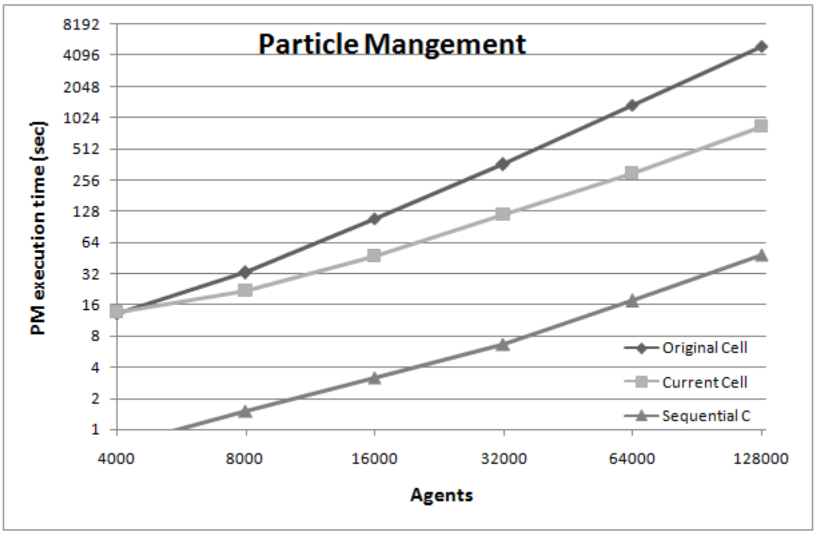


Figure 4.10: Particle Management run times of Cell and sequential C implementation.

Sequential Components

Following Amdahl's law (section 2.2.3), the overall parallel speedup is always limited by the run time of the slowest sequential component of an algorithm. In this chapter we will therefore analyze the impact of the sequential PPE processing functions on the overall performance. As discovered in section 4.5, the sequential overhead of the simulation still prevents a perfectly linear run time scalability, which originates from the non-linear scalability of sequential PPE processing. In order to overcome this limitation we therefore need to gain insight into the execution of the most significant sequential components of the simulation.

There are two significant sequential components running on the PPE, the Particle Management process and the Agent State Change loop. Both functions are concerned with agent management and include linear searches over the whole agent array. We therefore need to look at the underlying memory structure in conjunction with the hardware implementation of the Cell, in order to explain the observed performance.

5.1 Memory: AOS vs. SOA

Within the Stream Programming Paradigm there exist two types memory layouts that can be adopted to agent based simulations. In a Structure of Arrays (SOA) organization, we define one general agent structure and represent each agent field as an array of values to define multiple agents. Conceptually, this layout is used for vector processing on the SPEs, where four agents are kept as a collection of `vector float` fields. In main memory, however, agents are kept in an Array of Structures (AOS) layout, where each agent is held in an aligned continuous block of 16 `floats`, and organized in multiple arrays (see Figure 4.3 in section 4.1.1. Since a `float` requires 4 Bytes, each agent occupies 64B of memory.

Furthermore, there are two predominant types of data access within the sequential agent management routines. Both functions perform a linear search through individual agent arrays, in order to identify sets of agents which will be copied to a different memory location.

The agent copy process itself works on each field (`float`) of a single agents, hence it reads and writes continuous blocks of data. Since the PPE's L1 Data Cache (DCache) uses 128 Byte atomic cache lines, it will copy two consecutive agents on accessing one field on the first agent. Thus individual agent copies are efficient, compared to the SPEs, which access agents in bulks of four and have no automated cache hierarchy.

Furthermore, the PPE's 512KB L2 cache is *write-back*. This means it will attempt to perform all data writes on-chip and hold off writing the data to memory until the cache contents are flushed [1]. Hence, when writing several agents to a condensed part of an array, for example when copying all newly dead agents to the end of the dead array, all copied *Dead* agents will be written to memory in one bus transfer.

In contrast to single-agent copies, a full scan of the an agent-array causes a much more erratic memory access pattern. Both sequential processes search an entire agent array by looping over each agent exhaustively and examining one or two agent fields. Since this access pattern is non-aligned with the agent data it will therefore cause several cache misses within the PPE cache hierarchy. Each cache line within the L1 DCache consists of 2 agents, which will cause $arraylength/2$ cache misses. If the PPU does not find the next agent in the L1 cache it will look into the L2 cache. The L2 cache lines are also 128 Byte wide, but are treated as sets of 8 [1]. Thus we obtain $arraylength/16$ L2 cache misses which will trigger an instant data transfer from memory. Even though the bus has a high-bandwidth, it is designed for high throughput through buffering. The more erratic point-to-point data load into the L2 cache causes several small one-directional data transfers. Thus, if there is no predictive cache load, the search traversal will continuously be interrupted by bus transfers.

When analyzing the data access patterns of the sequential processing components we need to keep in mind that single-agent copy operations use the PPE's cache hierarchy very effectively, while exhaustive linear searches are costly and inefficient when using AOS agent layout in memory.

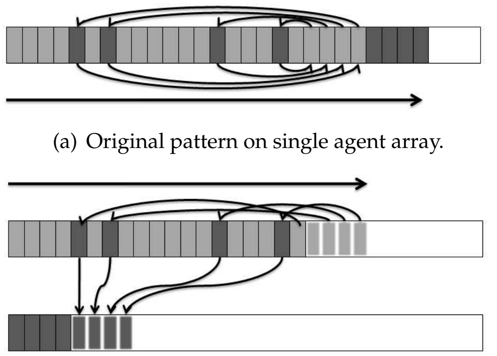
5.2 Agent State Change

The Agent State Change loop is a prime example of the type of data access use within the PPE components of the VEW simulation. The loop identifies all agents kept within the Living stage array, that changed their internal stage to Dead within the last iteration. That is to say that it looks for all newly dead Diatoms and copies them to the appropriate agent array. As such it is an artefact of the homogeneity criterion imposed by the Streaming paradigm on VEW simulation, and can be rated as a pure parallelization overhead.

In the previous chapter we have shown that, although linear in nature, this overhead is larger than the time taken for parallel updates itself. That is due to the fact that this function must be run before the Particle Management process, and is the first component running on the PPE which has to traverse the whole Living agent array during each iteration. We will therefore analyze the loop's structure and copy patterns and show a technique to significantly reduce this overhead, using SPE meta-data transfered via the implemented feedback loop.

5.2.1 Linear Search

This loop's implementation caused a severe performance bottleneck in the original Toymodel code. When all agents were kept in a single array, the loop would copy each dead agent it encounters to the end of the array and fill its space with a living agent. The one-pass traversal of the agent array, however, would ignore the offset calculated during the last iteration, causing it to apply the same technique to the block of dead agents already located at the end. This would cause a complete re-shuffle of all agents in the Dead section of the array, where each *Dead* agent switches place with another *Dead* one. This results in a non-linear increase in run time for larger amounts of agents.



(a) Original pattern on single agent array.

(b) New pattern traversing Living agents array in one pass.

Figure 5.1: Linear search based Agent State Change copy patterns.

With the introduction of stage-separated homogeneous agent arrays, the performance of this loop was reduced to a linearly increasing overhead. Due to array separation the unintended agent shuffle was avoided, and the search results in one complete pass over the Living agents. In the Toymodel, the *Living* agents far outnumber the *Dead* ones, due to the fact that the Particle Manager rules aim to merge all *Dead* agents into one agent per layer. Therefore the search complexity decreases only minimally with increasing agent counts, since the overall number of dead agents in the column is mainly constant.

The algorithm thus traverses the complete *Living* agents array, copying each *Dead* agent it finds to the end of the *Dead* agent array. At the same time it removes the agent by copying the last agent in the *Living* array into the vacated spot and decreasing the array length. It then checks the new agent's stage in this slot again before continuing its traversal.

During the search it inspects one field of each agent (the stage), resulting in the skipping pattern detailed in the last section. Thus it incurs regular cache misses, resulting in low utilization of the PPE cache hierarchy and irregular data load via the bus. This overhead, however, is limited by the number of Living agents per iteration.

5.2.2 Indexed Agent Copy

In order to reduce the overhead create by the linear search we introduced a technique to prevent the erratic search pattern. This was added the current version of the prototype and included the use of the Feedback loop implemented as part of the Farming DMA Framework. The idea is to prevent the PPE from traversing the complete agent array by gathering the required meta-information when performing the parallel agent update.

Since the SPEs convert each agent field into a vector at some point of the update iteration we can use branch-prediction to efficiently detect state changes on the vector level. The `teardown()` function returns a flag vector which gets transformed by the DMA loop into offsets within the processing block of agents. A dynamic array of intra-block offsets is then transfered to memory within a Feedback block, whose memory address into a synchronized to the corresponding Task Block. Due to Triple-Buffering no significant overhead is introduced.

Knowing the individual block offsets, the PPE can calculate indexes into the agent array for each Living agent that experienced a state change during the last iteration. Since the pre-calculated offsets are static, we have to defer the removal of dead agents withing the Living array until all of them are copied to the *Dead* array. Thus we require two passes at the *Living* array, but we do not have to inspect every Living agent.

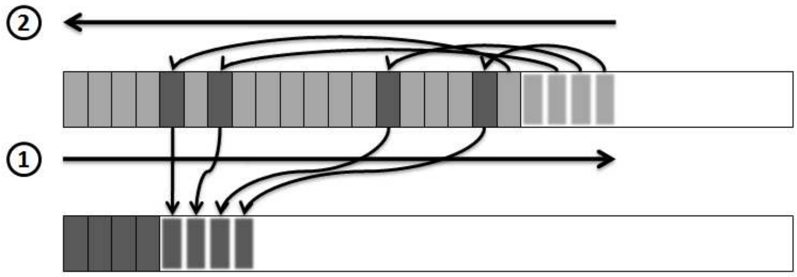


Figure 5.2: Indexed Agent State Change copy pattern without searching agent array.

This approach utilizes the PPEs cache hierarchy much more efficiently, since it only accesses individually targeted agents. Thus the 128 Byte cache line accessed for reading an agent contains all `floats` of this agent's state. Additionally, copying all newly dead agents to one continuous area of memory causes only one data write transfer on the bus, due to the *write-back* property of the PPE's L2 Cache.

The results of the implementation change are small when timing the overall run time performance, since for large simulations the Particle Management routine dominates the PPE processing. However, individual timing of the Agent State Change run time added over the full simulation (35040 iterations) shows the performance improvement gained by the technique (Figure 5.3(a)). It is important to note that the search loop used in the previous implementation is very similar to the searches used for Particle Management. When analyzing the PM function we need to keep in mind, however, that the PM requires several exhaustive linear searches over individual agent arrays.

Agents	Linear Search	Indexed Copy
4000	9.2	0.3
8000	15.8	0.9
16000	27.1	0.9
32000	53.3	1.5
64000	108.9	3
128000	243	6.1
256000	525.6	10.7
512000	1089.5 (18min)	21.1

(a) run times in seconds.

Figure 5.3: Indexed vs Linear Search copy loop.

5.3 Particle Management

As identified during previous investigations [5], the current implementation of the Particle Management is not applicable for parallel execution. Following Amdahl's Law the scalability of the overall algorithm is limited by the most costly sequential component. The PM presents such a bottleneck for any parallel VEW implementation. Although the PM is intended to provide a balance between model accuracy and computational cost, on the Cell architecture Particle Management becomes the dominant function within the algorithm. This can be seen when comparing the individual PM run time with the overall run time of the simulation in Figure 5.4.

In a general VEW simulation Particle Management is run after every iteration. This is not necessary for the Toymodel simulation, however, where running PM once every simulated day (48 timesteps) is sufficient to achieve the intended accuracy. For simulations with small numbers of agents this does yield a small performance increase. For large simulations, on the other hand, the PM's impact on performance remains.

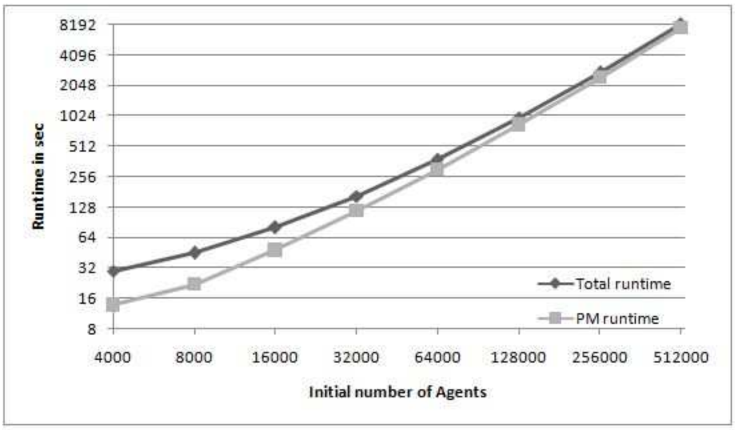


Figure 5.4: run times for exponentially increasing simulation sizes, showing dominance of Particle Management.

5.3.1 Split and Merge

Particle Management is intended to limit the overall number of agents simulated in the water column. It is designed to ensure that the number of agents in each column layer is within a specified range throughout the simulation. This balances model accuracy with computational cost.

When identifying a layer with an insufficient agent count the PM will split the largest agents in the layer until the minimum layer count is achieved. When splitting an agent the process creates a new agent with an independent trajectory. It copies the agents internal state and assigns half of the original subpopulation to each agent.

Similarly, when encountering a layer with too many agents, the PM will merge the two agents with the smallest subpopulation within the layer, until the agent count is below the specified maximum. The Merge function calculates a weighed average between the two agents for every field of the internal state, before adding the according sub-populations.

The PM uses model-specific rules to determine minimum and maximum boundaries. These rules are specified per agent type (stage) for sets of layer within the column, including the set of layer above the turbocline, which changes dynamically. The SPEs accumulate the numbers of agent of each type for each layer. Thus the PM knows the layers which need splitting or merging. However, identifying the individual agents to split/merge requires an exhaustive traversal of the agent array. This linear search loop is very similar to the one used for identifying agent state changes, in that it only inspects the agents current layer and sub-population, resulting in a non-linear memory access pattern.

The linear search pattern is applied once for every layer that needs splitting, in order to identify the set of largest agents within the layer. The Merge process, however, can require several searches per layer, since it can only ever merge two agents at once. Thus, when the number of necessary Merges is greater than half the number of agents in the layer, it will apply a Divide-And-Conquer method, halving the number of agents in the layer per invocation. This worst-case-merging mainly happens for *Dead* agents with a small search space. Thus the overall number of invocations of each PM function type scales linearly with the number of agents, as can be seen in Figure 5.5.

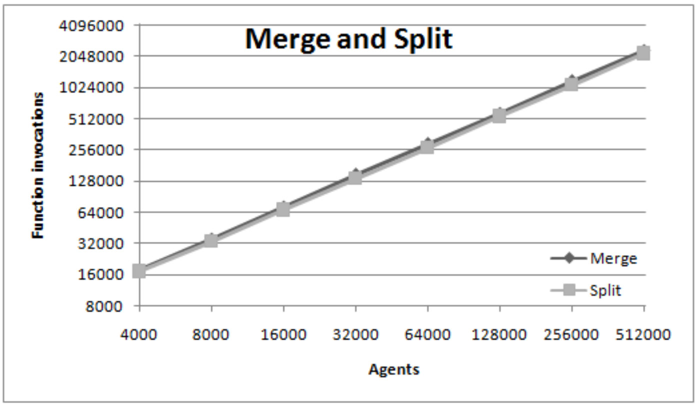


Figure 5.5: Total number of PM function invocations.

5.3.2 L2 Cache

The overall PM run time shows a non-linear scale factor, whereas there is a linear number of *Split* and *Merge* calls. The reason for non-linear performance curve must therefore be due to hardware features. Indeed, the large difference in PM run time between Cell and the x86 platform (Figure 4.10 in section 4.5) suggest a significant impact on PM performance due to the PPE's cache hierarchy and memory connection.

The main hardware limitation we identified was the size of the PPE's L2 cache. The PPE features a 512KB cache size, which corresponds to 8K agents of 64B. The x86 CPU used for comparison, on the other hand, has a 2MB associated L2 cache, which can hold up to 32K agents. The particular number of agents in the column, however, varies over the run of the simulation. Thus a simulation starting with 16000 agents spends a large number of iterations with less than 8K *Living* agents.

This means that all linear searches of the *Living* array can be done on-chip and the agent array only needs to be read once for the duration of PM processing. Similarly, due to the *write-back* property of the cache all agent copy movement will be written only once. If the simulation consistently contains more agents than the L2 cache can hold, the PPE will have to flush and re-load the cache for individual partitions within the agent array, resulting in several full cache reads and writes for each linear search traversal.

This effect can be shown by comparing the individual PM run times of Cell implementations with and without the Indexed Copy method detailed in the previous section. Figure 5.6 only shows a significant run time difference for simulation sizes below 16000 agents. This is due to the fact that the Agent State Change loop is executed before the PM routines, which causes it to load the complete agent array into the cache prior to PM execution. The trend of the pre-cached implementation, however, suggests PM execution time increases linearly with a steep gradient.

A similar increase in slope gradient can be observed for PM run on the x86 comparison architecture. Since simulation size and implementation of the PM code are equivalent between the sequential implementation and our Cell prototype, we can estimate equivalent PM workload between the the simulations. The sequential C implementation was run on a 1.6GHz dual-core CPU with 2MB L2 cache, which is equivalent to 32K agents. The predicted gradient increase for models with more than 32000 agents can be seen in Figure 5.7. It is important to note that the PM function executes much faster on the x86 architecture though, suggesting a far superior cache hierarchy and memory-bus connection.

Assuming that PM execution time progresses linearly after exceeding the L2 cache agent boundary, the curve in overall prototype performance (Figure 4.8 in section 4.5) can thus be interpreted as the PM gradient ”overtaking” the linear Update run time due to a steeper gradient increase caused by L2 Cache loads.

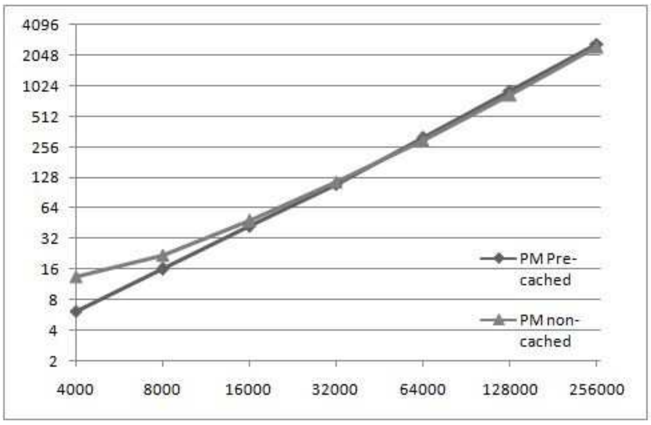


Figure 5.6: Difference in individual PM run times due to previously caching of the agent array.

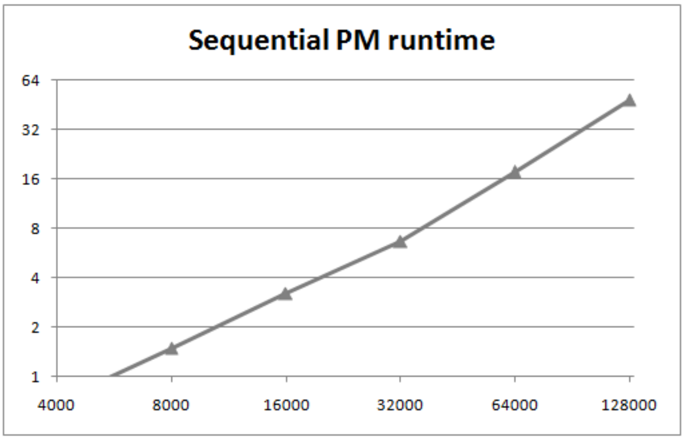


Figure 5.7: Sequential Particle Management execution time.

5.4 Component Evaluation

As shown in Figure 5.4, the Particle Management component dominates the overall execution time of the simulation for large numbers of agents. Thus, the changes applied to the Agent State Change loop in section 5.2 are barely noticeable on run time graphs. In this section we will therefore analyze the contribution of the individual sequential components discussed in this chapter towards the overall execution time, in comparison to the parallel update run time.

Figure 5.8 shows clearly how the linear Update component gradually diminishes with growing agent numbers. Eventually the agent copy and PM dominate the execution time.

The immediate effect of the indexed agent copy approach can be seen in Figure 5.9. This method shrinks the parallel overhead due to agent ordering to insignificance. The resulting graph reveals how the PM grows at a larger rate, gradually dominating the performance of the prototype for high agent counts.

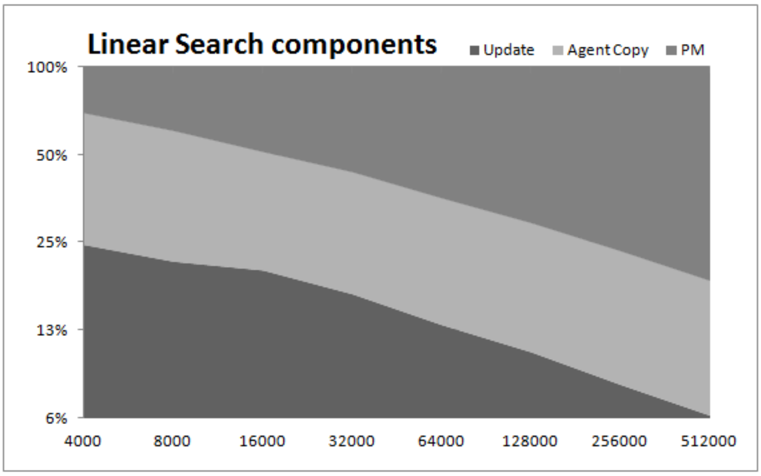


Figure 5.8: Percentage of run time components with Linear Search Agent State Change.

It is worth noting that the relation between Update code and PM run time remains constant from 4000 agent to 8000 agent models. After surpassing the L2 cache size equivalent, however, we can see how the PM component scales at a higher rate, causing increasing performance dominance.

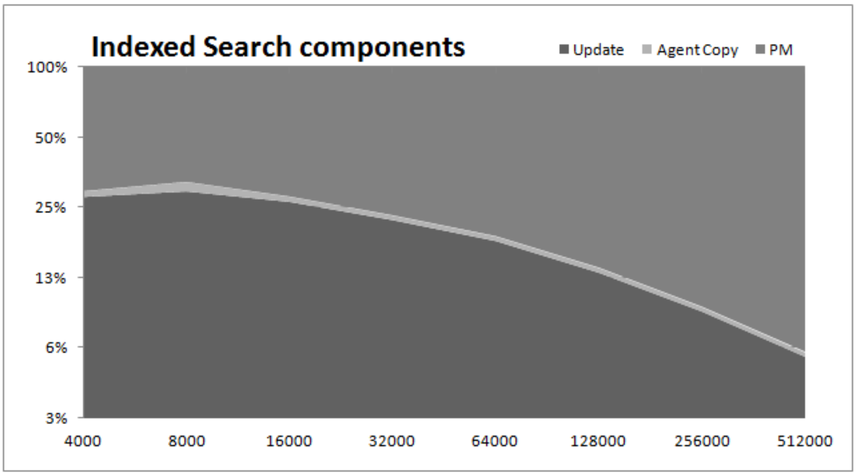


Figure 5.9: Percentage of run time components with Indexed Agent State Change.

Discussions

In order to evaluate the overall success of this project we first need to verify that none of the changes introduced into the prototype simulation compromised the correctness and accuracy of the simulated model. In this chapter we will therefore discuss the results obtained from different implementations of the Toymodel simulation and in order to estimate the value of the created prototype. For this we need to keep in mind that we cannot re-create identical simulation runs between sequential and parallel simulations due to parallel Random-Number-Generation. Thus we need to investigate the overall shape of the output graphs.

After detailing the main performance improvements achieved in this project in sections 4.5 and 5.4, we will furthermore evaluate the achieved parallel speedups and overall parallel scalability of the VEW algorithm on the Cell. This will give insight into the the performance advantages and limitations expected from Cell-based VEW simulations.

6.1 Model Correctness

In this section we analyze the results of different implementations of the Toymodel simulation used throughout this project. We aim to show, that none of the changes applied to the Cell implementation compromised the simulated output. This is a difficult task, since we cannot reproduce the exact results of a sequential implementation on a parallel platform, as described in section 4.1.2. The results of Java simulations were obtained on the 1.6GHz reference architecture detailed in section 4.5.

6.1.1 Biomass

The biomass of individual plankton species is the main evaluation criterion for VEW simulations. The expected behaviour of Diatoms is well known and documented in [6]. We therefore present the overall count of living Diatom cells in the virtual water columns. All simulations run during this project simulate a Diatom ecosystem simulated over the course of two years in a statically anchored water column in the Azores region. We sampled the biomass calculated by the simulation per simulated week.

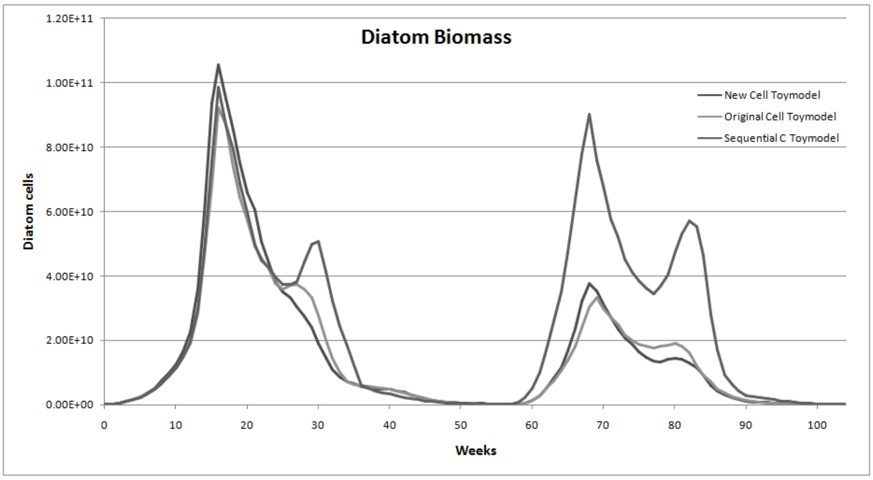


Figure 6.1: Living Diatom biomass simulated over 2 years by the Toymodel.

When analyzing Figure 6.1 we can see that our performance-optimized Cell implementation matches the results of the original Cell Toymodel quite closely. A perfect match cannot be expected, since we changed the Random-Number-Generator, resulting in an overall different simulation. One has to consider that all VEW simulations represent noisy systems. Thus a general error is included in any model, resulting in no two simulations producing identical outputs.

However, we can identify the characteristic shape of Diatom population growth. Two blooming periods are clearly visible during the summer months. A secondary peak can also be seen in the diagram, representing the autumn bloom of Phytoplankton. This bloom is generally smaller than during the summer period and is caused by the sunlight traveling to deeper layers of the column, which provide additional nutrients to the Diatoms.

Both Cell implementations produce results that significantly differ from the ones obtained by the sequential C implementation. Although the overall shape during the second year shows VEW characteristics, an overall decrease in population size has to be noted. This hints at a loss of nutrient chemicals, providing insufficient food for Diatoms to achieve the population sizes of the sequential implementation. It should also be noted that the Cell implementation of the Toymodel does not include a particular feature of the full-scale VEW arithmetic, which recycles nutrients that would drift out of the open bottom-layer of the water column during the course of the simulation.

Since this project is largely concerned with run time performance, we chose not to pursue further model debugging on the prototype in favour of the analysis of sequential components presented in chapter 5.

6.1.2 Agents

Throughout this project we have investigated the performance scalability of VEW simulations by increasing the initial number of agents in the column. This is based on the assumption that all implementation of the Toymodel show equivalent progressions of agent counts during one simulation. We verified this assumption on numerous occasions for all Cell implementations by summing the total number of individual Update function invocations. As expected these increased linearly with the increasing number of initial agents.

In Figure 6.2 we present the number of agents in the virtual water column over the course of a two year simulation for the most recent Cell prototype, as well as the sequential reference simulation code written in C. We can identify largely similar patterns for both models. The small differences exhibited can be attributed to the overall difference of the simulation detailed in the previous section. For the purpose of this investigation we can therefore conclude that the initial number of agents in any VEW model implementation gives an accurate indication of the overall simulation size, and hence the total work load of the parallel algorithm we are investigating.

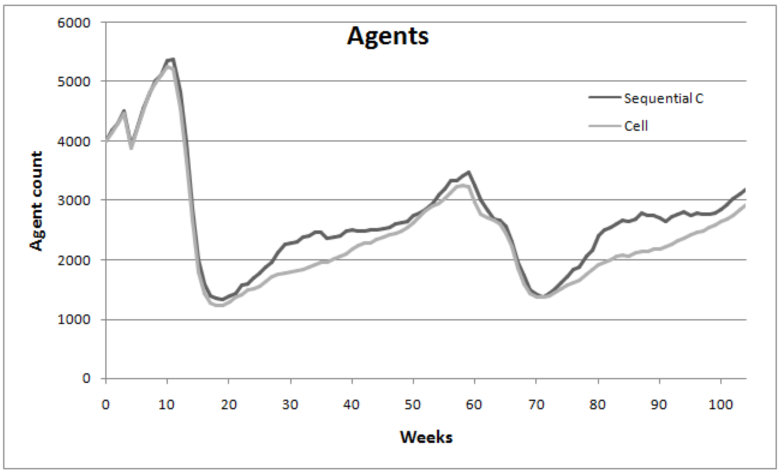


Figure 6.2: Number of agents in the virtual water column over two simulated years.

6.2 Performance

Several performance gains have been achieved throughout his project. In section 4.5 we discussed the improvements gained on Update code computation from parallel SPE vector processing, and showed how this scales linearly to large simulation sizes. However, this comes with an associated cost. In section 5.4 we detail the parallel overheads created by this parallelization and showed a technique for eliminating one costly search routine.

In this section we will now analyze the achieved parallel speedup of the vectorized Update code over the fastest known sequential implementation. This represents the only form of pure speedup due to parallelism in our prototype implementation. We will then highlight all the restrictions imposed by the current Cell hardware on the sequential parts of the code, in order to evaluate the Cell as a platform for large VEW simulations.

6.2.1 Speedup and Scalability

In section 4.5 we used a 1.6GHz x86 dual-core CPU as a reference platform for the Toymodel simulation, in order to match the speed of the slowest cell processing core. The Cell's fast SPE units used for parallel update computation run at 3.2GHz though. We therefore conducted a second series of sequential simulations on a different reference CPU clocked at 3.2GHz. This processor is a quad-core Intel Xeon x3350 with 12MB L2 cache and 2GB memory. It should be noted though that this machine uses faster DDR3 memory (compared to DDR2 used in the other reference setup), providing a quicker memory connection via the Front Side Bus.

The speedups detailed in Figures 6.3 and 6.4 are calculated from the individual Update run times of the Cell implementation compared to the Update execution of the sequential C Toymodel running on x86 platforms at clock speeds 1.6GHz and 3.2GHz. They most importantly show that our Cell prototype can sustain the achieved parallel efficiency for large numbers of agents, since none of the graphs shows a negative gradient.

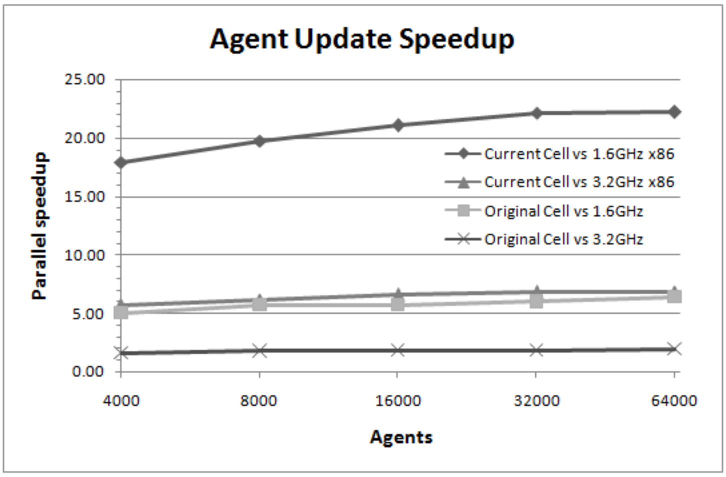


Figure 6.3: Parallel speedup of Cell Update code over x86 CPUs at 1.6GHz and 3.2GHz.

We can access only six of the Cell's eight SPEs, suggesting a maximum parallel speedup of six, since no update computation happens on the PPE. The gathered result series show a much larger gained speedup though (Cell vs. 1.6GHz). This is due to the fact that although we only use six processing cores, SIMD computation on vectors of four agents allows us to ideally process twenty-four agents at a time. Speedups of more than six can therefore be regarded as a criterion for efficiently utilizing vector computation.

And indeed we do achieve a speedup greater than six, even when compared to the 3.2GHz CPU (see Figure 6.4). It is important to keep in mind though, that both reference architectures employ a fundamentally different data and cache hierarchy. Thus we can only regard these results as indications into the efficiency of our vectorized Update code. However, running the Update for a 4000 agent simulation on only one SPE yields a speedup of 1.19 in comparison to the 3.2GHz model and 3.76 when compared to the 1.6GHz core. The general scalability of Update execution time with increasing numbers of processors, shown in Figure fig:perf-spe-scalability, indicates the expected decrease in execution time with added SPE processing units.

Agents	Current Cell		Original Cell	
	1.6GHz	3.2GHz	1.6GHz	3.2GHz
4000	17.91	5.65	5.01	1.58
8000	19.75	6.11	5.67	1.75
16000	21.05	6.53	5.70	1.77
32000	22.12	6.79	5.99	1.84
64000	22.17	6.18	6.34	1.95

Figure 6.4: Parallel speedup of Cell Update code over x86 CPUs at 1.6GHz and 3.2GHz.

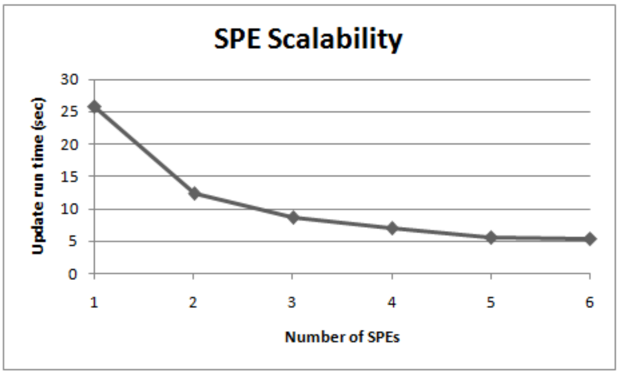


Figure 6.5: Parallel speedup of Cell Update code over x86 CPUs at 1.6GHz and 3.2GHz.

6.2.2 Limitations

The greatest limitations imposed on parallel Cell-based VEW simulations stem from the sequential overhead of linear agent-array scans, as used for Particle Management. Amdahl's Law implies that this imposes a limit on execution scalability of the VEW algorithm for parallel platforms. Comparison with x86 reference CPUs suggests, however, that the overhead stems from particular hardware restrictions imposed by the PPE cache hierarchy.

As described in section 5.3 we suggest that PM execution scales linearly. The gradient of linear run time increase, however, changes for simulations containing agent arrays with more agents than the L2 cache can hold. Thus, for models simulating large numbers of agents PM execution grows faster than Update computation. This results in the curved performance graph observed in Figure 6.6. The diagram also shows the performance gain of our Cell code over the previous Toymodel implementation as well as the sequential code running on the fast CPU. The curve suggests, however, that this parallel efficiency cannot be maintained for larger simulations.

On the other hand, due to the elementary linear nature of PM processing we can predict, that increasing the model size through addition of new agent-stage arrays will result in an overall linear performance scalability. The Toymodel has an inherently uneven distribution of agents between arrays, caused by the fact that we merge all *Dead* agents within a layer into one. From full scale VEW

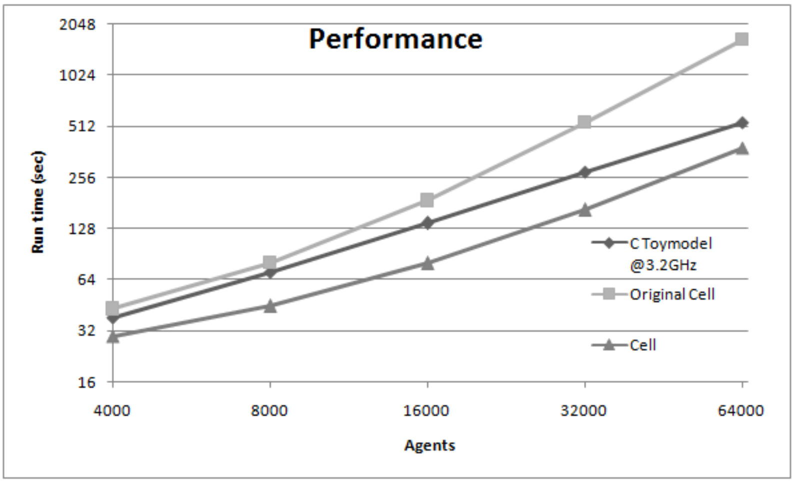


Figure 6.6: Execution times of Toymodel implementations on Cell and in comparison to sequential C code run at 3.2GHz.

simulations we can expect a generally more even distribution of agents among stages. Thus adding several plankton species should not limit performance scalability of small simulations, as long as the new arrays themselves do not grow too big for the cache.

We also have shown how to parallelize a PPE search loop in section 5.2.2. This method demonstrates an efficient utilization of the different cache hierarchies of the Cell cores. The approach use the PPE caching system to executed the agent copy operations required by the algorithm. Since linear search scans on the PPE create the most significant performance bottleneck, we can use the fact that the SPE-based Update loop traverses each agent during an iteration. By creating a condensed representation of agent meta-data we can thus guide the PPE copy operation and reduce the search space of sequential agent scans.

Completely re-structuring the Particle Management, however, can have significant influences on the results of VEW simulations. In order to facilitate a parallel PM search algorithm we would require to change the global *Split/Merge* rules to work on subset of agents created by the allocation of agents to SPE units. Thus, a parallel PM strategy can only impose local boundaries on the number of agents per processing core. An overall probabilistic approach to Particle Management, however, is beyond the scope of this project, since the impact on model correctness has to be evaluated thoroughly.

One problem with linear PM searches is that they only expect a limited set of agent variables which results in irregular memory access patterns in AOS agent layout. With the SPU-based agent update locally converting the agent data into SOA, the possibility of a general SOA layout in memory has to be considered as a potential future option. The current DMA strategy, however, prevents this change. Due to the limited local buffer size, we need to use a block-based agent distribution. Therefore, an additional memory buffering scheme would be needed in order to accumulate agents from SOA-based agent arrays into DMA-ready blocks. We would require additional synchronization between PPE and SPEs in order to facilitate this buffering.

Conclusions

The aim of this project was to investigate the utilization of Stream Processing on the Cell platform for parallel computation of VEW simulations. We have succeeded in demonstrating that the main work load of computational Lagrangian Ensemble models due to agent updates can efficiently be speed up using the Cell's SPE Vector Processing units. The Agent Update speedups achieved over sequential implementations are maintained for simulations with increasing numbers of agents, resulting in linear parallel scalability of Update run time. We furthermore provide a prototype Model Compiler that is able to create vectorized update code which uses the SPE execution kernel developed in this project.

We also analyzed the limitations of the Cell processor imposed on VEW models by sequential processing components. These cause a non-linear increase in run time for models with very large numbers of agents. We showed how the PPE's L2 cache limits the performance of the Particle Management function and causes it to dominate execution for large models. Following Amdahl's Law of parallel computation this imposes an upper limit on the overall performance gain achievable for Cell-based VEW simulations.

7.1 Parallel Scalability of VEW models

During this project we have shown that a significant parallel speedup can be achieved for Agent Update computation. In chapter 4 we demonstrate an execution framework for fully vectorized SIMD calculations using Stream Programming methods. This prototype framework maintains high parallel efficiency in a linear fashion when scaled to large numbers of agents. When compared to a sequential C implementation run at 1.6GHz we achieved Update speedups greater than 22 (see Figure 6.3 in chapter 6), as well as a significant overall run time improvement for simulations with more than 100,000 agents (see Figure 4.8). The largest model run on the final Cell prototype simulated an initial population of 1,024,000 Diatom agents over 2 years in 5 hours.

Furthermore we identified Particle Management as the remaining bottleneck restricting perfectly linear scalability of execution times. Parallel speedups obtained from Update computation are sufficient to achieve performance gains for reasonably sized simulations. This is countered, however, by a growing dominance of PM execution.

In chapter 5 we identify hardware limitations in the PPE cache hierarchy as the reason for the dominance of sequential agent processing. The PPE caching system limits the performance of linear agent search scans, in contrast to the efficient use of the Cell's bus for high-bandwidth data transfer to the SPE units.

We were not able to overcome the sequential overhead created by Particle Management. This is due to the strong impact of the underlying algorithm on the models accuracy. From our findings we can conjecture, however, that a Particle Management process based on parallel SPE-based searches will decrease the sequential overhead significantly and provide a very fast and efficient solution to multi-core VEW simulation. We support this claim by demonstrating the use of a SPE-based parallel search in conjunction with agent copy executed on the PPE in section 5.2.2. This method significantly reduce the parallel overhead incurred from the stage-homogeneity requirement for agents established in chapter 4.

Throughout the project we also gained valuable insights into Stream Processing for agent-based simulations. We established that a memory structure holding agents in multiple stage-homogeneous arrays (described in section 4.3) is superior to a single-array approach on the Cell. This intuition is useful to consider for further attempts at parallelizing VEW simulations on multi-core platforms, since it originates from the natural separation of Update functions by an agent's biological stage.

7.2 Update Code Generator

In addition to performance gains we also demonstrate a partial Model Compiler that is compatible with the Stream Processing structure developed for the Toymodel prototype. This code generator is able to create vectorized Update arithmetic for any type of plankton agent from the standard model description files used in the VEW. This forms a basis for future integration of performance-oriented platform-specific model compilation for the VEW applications.

We propose separate `startUpdate()` and `endUpdate()` functions, as used in the current Java VEW, to decouple the Update arithmetic from SPU kernel code. This allows for consistent use of vectorized SPU Update functions. It also allows for generic environment requests to be gathered and passed between agents and the virtual water column without significant overhead. This is a key property for the additional feature integration needed to simulate full VEW models.

7.3 Future Work

Additional features, such as predatorial ingestion, still need to be added manually to the code generator. Using the same approach of padded vector conversion as presented in section 4.1.3, this will not compromise the parallel Update efficiency of the Vector Processing structure of the Toymodel prototype. The provided Model Compiler is easily extensible by using the developed prototype simulation as a template. This, however, requires a more efficient implementation of Particle Management.

7.3.1 Ingestion

One main feature of VEW simulations is the ability to model Predation, where Zooplankton feeds on the Phytoplankton population in the virtual water column. This represents one of the most complex agent-environment loops in the current VEW implementation. Ingestion requests have to be accumulated per column layer and scaled according to the availability of food. This process is very similar to nutrient uptake and release, which is already part of the Toymodel simulation. We can therefore conjecture that the addition of this functionality can be implemented using the same methods described in this report.

Several other features, such as Chemical Budgeting and Recycling functions also need to be added to a Cell-based Model Compiler before we can attempt to reproduce the results of the fully working Java implementation.

7.3.2 Parallel PM

Particle Management was identified as the main performance bottleneck in this report. Several approaches to optimizing this sequential overhead need to be considered. Following the findings of chapter 5, we propose to use parallelized agent search routines on the SPEs. These can be implemented similar to the detection of Agent State Changes and will reduce the search space for PPE-based agent management.

This, however, implies a re-structuring of the general PM algorithm as applied to Lagrangian Ensemble models. For parallel PM searches on several nodes, the traditional min/max boundaries on agent numbers can only be applied locally. This results in a probabilistic PM algorithm which cannot define hard boundaries on a model's size. Thus parallel PM needs a thorough investigation that analyses the potential impact of this change.

Bibliography

[1] M. Scarpino. *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 1st Edition, 2008.

[2] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[3] J. D. Woods. *The Lagrangian Ensemble metamodel for simulating plankton ecosystems*. Progress in Oceanography 67, 2005, 84-159

[4] W. R. Hinsley. *Planktonica: A system for doing biological oceanography by computer*. PhD Thesis, Department of Computing, Imperial College, London

[5] T. Field, W. Hinsley, W. Kleiminger, P. Meier, J. Wright, K. Bijlani. *2008 Summer UROP Project*. Department of Computing, Imperial College, London

[6] M. Sinerchia. *Testing theories on fisheries recruitment*. PhD Thesis, Department of Earth Science and Engineering, Imperial College, London

[7] W. R. Hinsley, A. J. Field, J. D. Woods. *Creating Individual-based Models of the Plankton Ecosystem*. International Conference on Computational Science, 2007, Lectures in Computer Science, Vol. 4487, 111-118. Springer.

[8] D. A. Bader, A. Chandramowlishwaran, V.Agarwal. *On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis*. 37th International Conference on Parallel Processing, 2008, 520 - 527.

[9] J. Lamoureux, T. Field, W. Luk. *Accelerating a Virtual Ecology Model with FPGAs*. 20th International Conference on Application-specific Systems, Architectures and Processors, 2009.

[10] T. Chen, R. Raghavan, J. Dale, E. Iwata. *Cell Broadband Engine Architecture and its first implementation*. IBM Technical Library, http://www.ibm.com/developerworks/power/library/pa-cellperf/, 2005.

[11] J. Gummaraju, J. Coburn, Y. Turner, M. Rosenblum. *Streamware: Programming General-Purpose Multicore Processors Using Streams*. ASPLOS, 2008.

[12] J. Gummaraju, M. Rosenblum. *Stream Programming on General-Purpose Processors*. International Symposium on Microarchitecture, 2005.

[13] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, W. J. Dally. *Executing Irregular Scientific Applications on Stream Architectures*. ACM, 2007.