

A Model for Programming Large-Scale Configurable Computing Applications

Carl Ebeling*, Scott Hauck†, Corey Olson†, Maria Kim†, Cooper Clausen*†, Boris Kogon*

*Dept. of Computer Science and Engineering and †Dept. of Electrical Engineering
University of Washington
Seattle, WA 98195

Abstract—It is clear that Exascale computing will require alternative computing substrates such as FPGAs as an adjunct to traditional processors to stay within power consumption constraints. Executing applications, or parts of applications, using specialized, fine-grained computing structures configured into FPGAs can achieve a large increase in both performance and energy-efficiency for many important applications. Unfortunately, the process of designing and implementing these specialized hardware structures is tedious and requires expertise in hardware design. This lack of programming models and associated compilers for configurable computing has impeded progress in the use of FPGAs in large-scale computing platforms. This paper presents a parallel programming model and compilation strategy that programmers can use to describe large-scale applications using familiar concepts and abstractions, but which can be implemented using large-scale, fine-grained parallel computing structures. We outline this model and then use it to develop a scalable parallel solution to the short-read reassembly problem in genomics.

INTRODUCTION

Our goal is to enable programmers to write parallel applications for large-scale computing systems that include a substantial amount of computing power in the form of FPGA accelerators. What we have in mind as an eventual target are platforms with 10's to 100's of processors and 100's to 1000's of FPGAs. Our approach is to adopt and adapt a set of techniques and concepts that have been developed for programming parallel and distributed computers to solving the problem of implementing large scale algorithms using hardware-accelerated computing platforms. This includes providing a computation model that enables algorithms to be described at a relatively high level of abstraction and then mapped across a large number of computation nodes that include processors, FPGAs, and possibly other accelerators like GPUs. This will allow applications to be partitioned and mapped to a combination of software and hardware resources using a single, consistent programming model.

Our model and tools will allow a designer to express the implementation of an algorithm as a set of communicating objects. Each object comprises data and threads of

computation that operate on that data, and interact with other objects via a buffered, flow controlled protocol. A separate system-level description defines how data arrays are partitioned and replicated to achieve high computational capacity and data bandwidth, how modules are duplicated and distributed to achieve parallelism, and how data and computation are assigned to CPUs, FPGAs and possibly GPUs. Computations that are co-located are connected together with efficient, local communication links, while elements on different chips, or those requiring unpredictable communication patterns, communicate via intra- and inter-chip communication networks. In this way, a designer can express the overall computation in a manner most appropriate to the problem being solved, and then map the design to the hardware to optimize the resulting implementation. Our goal is to provide mechanisms to help automate this mapping, as well as communication synthesis approaches to optimize the networks to the actual demands of the computation.

PROGRAMMING MODEL

Our model is based on an object model: A system is comprised of objects, each containing data and methods that implement the functionality of the object. Our research is focused on implementing objects in configurable hardware (FPGAs), and compiling programs into large circuits. However, the objects in our model can also be implemented in software and mapped to a processor in the system. This allows a system to be built using a combination of hardware and software, and functionality can be moved between hardware and software by specifying whether an object is implemented in a processor or an FPGA.

An object generally has state, that is, data that it owns and manages, which may be in registers or in privately owned memories. Objects interact via methods, which implement the functionality of the object. All objects have a main method, which is started when the system starts. Main methods may simply initialize the object's data and then exit, or it may continue to

run as an independent thread in the style of a main program. Objects call each other's methods, which may be either asynchronous or synchronous: Asynchronous methods, which are the default, do not return values, and the caller can immediately continue execution after calling an asynchronous method. A synchronous method call requires the caller to wait for the called method to complete, generally with a return value.

All hardware objects in a system are static, meaning they are created and connected together at compile time. This is done using constructor methods, which use meta-programming constructs and compile-time information to construct the system hierarchically. This is essentially identical to many generator-style programming languages[1], as well as structural descriptions and generators in Verilog.

Methods are implemented as independent hardware threads that wait to be invoked. This means that all the methods of an object run in parallel, using arbiters (locks) when necessary to make sure that shared object resources are used in a consistent way. Method calls can be implemented in a number of ways depending on the location and type of the objects that are communicating. The most general mechanism uses messages over a network to implement method calls as remote procedure calls[2], [3]. Remote procedure calls support dynamic method calls, where the callee object is dynamically specified. However, RPC incurs substantial unnecessary overhead for objects that are located close together and can interact via a direct wired connection. In such cases, the method call can be implemented using a set of wires that implement either a synchronous or asynchronous method call, reserving the general network for non-local communication and dynamic method calls.

Dynamic Objects

Although hardware objects are static in that they are constructed before the system begins execution, dynamic allocation and deallocation of objects is a convenient abstraction that reflects how objects are often used. This pattern is quite common where an object is used temporarily for a specific use, where a dynamic number of instances of the object are needed at any one time, and where a large number of instances are used over the lifetime of the computation. As we will see, dynamic objects can also finesse the problem of synchronization in many cases.

Dynamic objects are implemented as a pool of statically instantiated objects managed by an object manager. The manager maintains a map that keeps track of free and allocated objects. Dynamic objects are differentiated by an object ID that is computed from a parameter, or

set of parameters, that are shared by all the object's methods. The programmer provides a map function from this parameter, or parameters, to the object ID. The first time that an object method is called with a particular ID, the corresponding object is automatically allocated by the manager, and initialization code, if any, is executed. Thereafter all method calls that map to the same ID are delivered by the object manager to this object. When the object has completed, it deallocates itself by informing the object manager.

Parallelization via Object Duplication and Distribution

In most cases, an application can be described using a relatively small number of communicating objects. Generating a parallel implementation involves making many copies of these objects in order to divide up large data sets and allow many computations to proceed in parallel. We borrow the idea of "Distributions" from parallel programming languages like ZPL[4], Chapel[5] and X10[6] to describe large parallel implementations. Distributions allow the partitioning and parallelization of an implementation across a large number of nodes to be described concisely and implemented automatically. Changing the parameters and even the shape of a parallel implementation is thus straightforward.

An object is duplicated and distributed using a distribution function that maps a method parameter, or a set of method parameters, to a "locale". A locale can be a specific FPGA or perhaps a part of an FPGA. The distribution function explicitly partitions the parameter space among a set of locales, each of which has a copy of the object. Where arrays are distributed, the distribution function also describes how the arrays are partitioned across the locales.

The programmer describes a parallel implementation simply by defining the distribution map functions and the compiler then automatically partitions the memory, duplicates the objects, implements method calls using the most efficient means, and routes method calls to the appropriate objects. A parallel implementation can then be tuned by redefining the distributions and recompiling.

To achieve even greater parallelism, memory objects can be replicated. For example, in a very large system, access to memory that is widely distributed may cause the network to be the bottleneck. This can be avoided by replicating the memory in addition to distributing it. Method calls are then routed to the nearest replica, reducing the network traffic.

In highly concurrent systems, almost all objects are distributed. Designing a distributed parallel system requires understanding the implications of a particular distribution on memory and communication bandwidth.



Fig. 1. Reassembly aligns each read to the correct position in the reference sequence. Sufficient redundant reads are generated to cover the reference sequence up to some read depth D .

Object distributions should be “aligned”, that is, distributed so that most communication between the distributed objects is local, ensuring that the only non-local communication used is that which is essential to the computation. With our model, the programmer can focus on describing the functionality of a system in terms of individual object classes separate from describing how those objects are partitioned and duplicated to create a parallel system. A system can be remapped using a different distribution very quickly since the changes required are automatically generated along with all the communication, including the allocation of method call invocation to local wires or the general network.

EXAMPLE: RESEQUENCING AND SHORT READ REASSEMBLY

Next generation sequencing technologies have appeared in recent years that are completely changing the way genome sequencing is done. New platforms like the Solexa/Illumina and SOLiD can sequence one billion base pairs in the matter of days. However, the computational cost of accurately re-assembling the data produced by these new sequencing machines into a complete genome is high and threatens to overwhelm the cost of generating the data[7]. Providing a low-cost, low-power, high-performance solution to the re-sequencing problem has the potential to make routine the sequencing of individual genomes[8].

Next generation sequencing machines “read” the base pair sequence for short subsequences of the genome, on the order of 30-70 base-pairs. These are called “short reads”, and sequencing machines can generate them in parallel by the hundreds of millions. To simplify somewhat, a long DNA sequence is sliced at random into many small subsequences, which are all read in parallel. By doing this for several copies of the DNA sequence, many overlapping reads are produced. This guarantees coverage of the entire reference sequence and also allows a consensus to be achieved in the presence of read errors. The goal of reassembly then is to align each short read to the reference sequence as shown in Figure 1. Reassembling these short reads into the original sequence is called the “short read reassembly problem”.

Algorithm Description

Our algorithm operates in two steps. In the first step, each read is looked up in an index to find the set of “candidate alignment locations” (CALs). These are the locations in the full genome where the read may match. We assume that this index is constructed offline since only one such index is needed for a reference genome. In the second step, a full Smith-Waterman[9] style algorithm is used to compare the read to the reference at each location in the genome, and the top scores are reported to the host for further processing.

We assume that each read maps to a specific location in the genome and our task is to find that location. Since the genome we are trying to reconstruct is different from the reference genome, and there are errors in the reads themselves caused by the readout process, we cannot use a simple index to look up the reads in the reference. Instead we use a “seed” approach[10] to construct an “approximate” index table. This approach assumes that there is a region in a read, called a seed, that exactly matches the corresponding region in the reference to which the read aligns. For example, we might use all 10-character contiguous subsequences of a 50-character read, hoping that at least one matches exactly to the correct alignment in the reference. The Reference Index Table maps each possible seed to all the locations in the Reference where that seed occurs.

Our algorithm is briefly described in 2. Each block represents one object, which includes the data and methods defined for that object. The arrows are used to show the method call graph. Synchronous method calls are shown with arrows in both directions.

a) Reads: The Reads object contains the array of short reads which are assumed to be in large DRAM attached to the FPGAs. The processReads() method is called by the Host to initiate the execution of the algorithm. processReads() reads the short reads from memory and sends them one at a time to the Seeds object by calling Seeds.nextRead().

b) Seeds: The nextRead() method takes the short read passed as a parameter, extracts all the seeds, and calls the RIT.nextSeed() method with each seed. Before it does this, it calls Filter.nextRead() with the read and

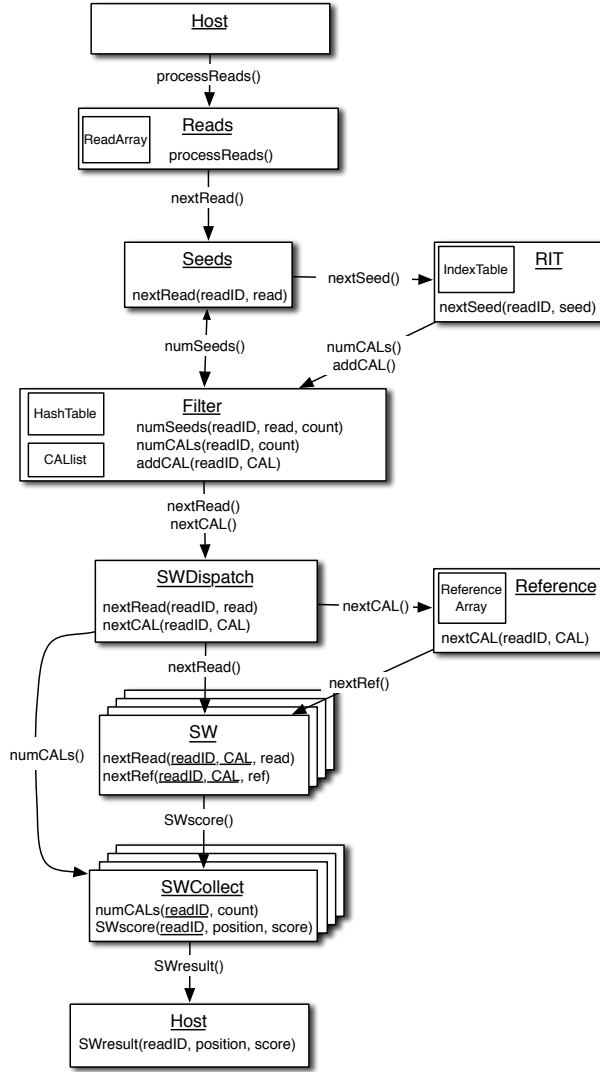


Fig. 2. This call graph shows the algorithm used to solve the short read reassembly problem in this example. For each read, a mask is applied to every possible position along the read to generate a seed. The seed is used as an address in a reference index to find a set of candidate alignment locations (CALs). The CALs are used for performing fully gapped local alignment on the reads at the specified location in the reference.

the number of seeds to expect. This is a synchronous call, so the `Seeds:nextRead()` method will block until the Filter object is ready for the next read. This is a barrier synchronization which reduces the amount of parallelism that is achieved at this point in the pipeline, but we assume that this does not affect the overall performance as the bottleneck is in the Smith-Waterman step, which occurs later in the process.

c) *RIT*: The RIT object owns the index table. When the `nextSeed()` method is invoked, it looks up the seed in

the index table which contains the list of CALs for that seed. It first calls `Filter:numCALs()` with the number of CALs that will be sent for the seed. It then calls the `Filter:addCAL()` with each of the CALs.

d) *Filter*: The purpose of the Filter object is to collect all the CALs for a short read, and pass along the unique CALs for comparison. The `nextRead()` method is called by Seeds with each new short read. This is a synchronous method call which waits until the previous read, if any, has been completely processed. It then initializes the Filter object for the new read, and returns, allowing the `Seeds:nextRead()` method to start generating the seeds for the read. This synchronization is required since Filter can only process a single short read at a time.

Each call to `addCAL()` inserts a CAL into the hash table and the CAL list, if it is not already there, or increments the count in the table if it is. When the last `addCAL()` has been received, `addCAL` then processes the hash table. It first calls `SWDispatch:nextRead()` to give it the next read, and then it goes through the list of unique CALs, each time calling `SWDispatch:nextCAL()`. It also clears the hash table entry for each, reinitializing the hash table. The Filter object knows when the last `addCAL()` method call has been received for a read by counting the number of seeds (`numCALs()` method calls) and the number of CALs per seed (total count in the `numCALs()` method calls).

e) *SWDispatch*: The `SWDispatch` object is essentially a front-end to the SW objects. It generates the appropriate method calls to initialize and begin execution of each SW objects. Each call to `nextCAL()` makes one call to `SW.nextRead()`, passing the short read to an SW object, and one call to `Reference.nextCAL()`, causing the corresponding call to `SW.nextRef()` with the reference string.

f) *Reference*: This object contains the reference genome data. When given a CAL via the `nextCAL()` call, it reads the section of the reference at that CAL and call `SW.nextRef()` to send it to the SW object.

g) *SW*: The SW object is a dynamic object, indicated by the multiple boxes in the figure, which means there are many static instances of the object that are managed by an object manager. The key for this dynamic object is the (readID, CAL) pair. The first method call with a new pair of values causes an SW object to be allocated and initialized. Two method calls are made to SW, one with the short read and the other with the reference data. When both methods have been invoked, the SW object performs the comparison of the short read to the reference section at the CAL location, and when done calls `SWCollect:SWScore()` with the result and deallocates itself. SW uses a deeply pipelined

dynamic programming algorithm, which means that it can actually deallocate itself *before* it has completed the comparison, but when it is ready to start performing a new comparison, which can be overlapped with the previous comparison.

h) SWCollect: This object collects the scores reported for all the different CALs for a short read. This is also a dynamic object which is keyed by the readID parameter. SWScore keeps track of the best scores reported, and when all scores have been reported, it calls Host:SWResult() with the best results. SWCollect also uses reference counting for synchronization: SWDispatch calls numCALs(), and when this many scores have been reported, SWScore reports the scores and deallocates itself.

Synchronization

It is worth noting that there are two types of synchronization used in this example. First, there is an explicit barrier synchronization used in the first part of the pipeline, whereby the Seeds object must wait for the previous read to be completely processed by the Filter object before it starts processing the next read. This ensures that the short reads pass through the objects in order. That is, all method calls associated with one read are made before method calls for the next read are made. This of course reduces parallelism. The second type of synchronization is enabled by dynamic objects and reference counting. These objects are allocated implicitly and method calls can be interleaved arbitrarily since they are automatically directed to the appropriate object. Reference counting allows objects to automatically deallocate when all outstanding method calls have been received.

Parallel Implementation

There is already substantial parallelism in the implementation as described, particularly with the many concurrent, pipelined Smith-Waterman units which are enabled by dynamic objects. To increase the performance of the implementation, we need to consider where the bottlenecks occur. If we assume that we can always make the pool of SW objects larger, then the bottleneck occurs at the RIT and Filter objects. For each read, Seeds makes a large number of calls to RIT:nextSeed(), each of which is a random access into the index table. If this table is stored in DRAM, there will be substantial overhead in performing this access. We can increase the performance of the RIT by partitioning it so that multiple accesses can proceed in parallel. This is done simply by describing a distribution function for the RIT that partitions the table.

This now moves the bottleneck to the Filter object, which will get multiple simultaneous addCAL() method calls from the RIT objects. We can remove this bottleneck by duplicating the Seeds, Filter and SWDispatch objects using an “aligned” distribution. That is, we duplicate these objects so that each Seeds object communicates with one Filter object, which communicates with one SWDispatch object, all of which are handling the same read. Assuming that reads are processed in order by readID, using the low-order bits of the readID to do the distribution uses these objects in round-robin order.

At this point, the Reference object becomes the bottleneck as several SWDispatch objects call Reference:nextCAL() concurrently. This can be solved by distributing the Reference object and partitioning the Reference memory into blocks. Assuming that the CALs are spread out more-or-less uniformly, this enables multiple concurrent accesses to proceed concurrently. We finally reach the point where the Reads object becomes the bottleneck. That is, we can now process short reads as fast as we can read them from memory.

Of course, we have described this implementation using an idealized view of the hardware platform. For example, there will be only a small number of large memories connected to an FPGA, and thus partitioning the RIT and the Reference into many memories for concurrent access can only be done by spreading the implementation across a large number of FPGA nodes. One option is to distribute the “main” objects (all except for RIT and Reference) across the FPGA nodes using an aligned distribution based on readID. This keeps the method calls between them local to an FPGA. However, the method calls to the RIT and Reference are non-local because neither of them can be partitioned by readID. Beyond some degree of parallelism, the communication implied by these calls will become the bottleneck since it is all-to-all communication.

An alternative way to distribute the objects is to use a distribution based on CALs. This partitions the Reference and the RIT, but it means that the remaining “main” objects are replicated instead of distributed. That is, each short read is processed by all of the replicated objects, so that the Seeds:nextRead() method call is broadcast to all the replicated copies of Seeds and remaining method calls are made on the local object copy.

Distributing/replicating by CAL means that each RIT object only has the CALs assigned to its partition and so the objects only handle a subset of all the CALs. This means that all of the communication between objects is local and so there is no inter-FPGA communication except for the initial broadcast. However, this partition-

ing has some consequence both on the algorithm and the performance. First, since each Filter object sees only a subset of the CALs, it cannot filter CALs based on information about all the CALs for a read, and so we may have to process more CALs than necessary. Second, if we partition the objects too finely, then many partitions may have no CALs at all for a read, and the objects will spend a lot of time doing nothing. So, for example, if there are an average of 32 CALs per read, we will start to lose efficiency if we partition much beyond a factor of 8.

We can, however, easily combine both types of partitioning. For example, we could partition and replicate by 8 first, and then within each partition distribute objects by another factor of 8. This partitions each RIT and Reference (by readID) by only a factor of 8, and the resulting communication bandwidth does not overwhelm the network.

Flow Control

In a system where data flows through a pipeline, work will pile up behind any bottlenecks. While in a simple pipeline where method calls are implemented using point-to-point wires and FIFOs, objects in the early part of the pipeline will be stalled and the pipeline will simply operate at the speed of the slowest object. In a more complex system comprising many distributed objects and network communication, the work that piles up can clog network buffers, which can affect other flows. The question is how best to throttle the rate at which work is created so this does not happen. In our example, work is created by the Reads objects and simple credit-based flow control can be used to throttle the rate at which it calls `Seeds:nextRead()`. Reads flow into the system at the top, and out of the system at the bottom when `SWCollect` reports the scores for a read. If we add a method `Reads:done()` which is called by `SWCollect` each time it complete reporting the scores for a read, then the Reads object can keep track of the number of reads in the pipeline. We can compute how many reads can be in the pipeline before the network might be affected, and allow Reads to only have this many reads outstanding in the pipeline. In complex systems, this number may be difficult to ascertain, although a safe number is the size of the smallest call FIFO. The best scenario would be for the system compiler to analyze the data flow and insert the flow control methods and work metering automatically.

CONCLUSION

This new object model allows programmers to quickly design new applications targeting large-scale computing

systems by abstracting the physical design and by moving the decision for physical location of objects from design time to compile time. This removes the burden of much of the physical design from the designer, including non-critical objects and communication infrastructure, but still allows the designer to use hand-tuned HDL in the system. The object model allows hardware objects to first be developed and easily debugged as software objects before implementing them in the target system. This enables the designer to rapidly develop and debug a system, thereby greatly reducing design cycle time. This model also has no need to declare the location of an object at design time, so objects can be partitioned and distributed across a large system without any required changes to the object description. With these benefits, a designer will more rapidly be able to design, develop, and debug an application for a large-scale computing system.

Acknowledgements

We would like to acknowledge Pico Computing and the Washington Technology Center for their support of our research.

REFERENCES

- [1] P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems," April 1998, pp. 175–184.
- [2] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.
- [3] J. Maasen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programming," *ACM Transactions on Programming Language Systems*, vol. 23, no. 6, pp. 747–775, 2001.
- [4] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, "ZPL: A Machine-Independent Programming Language For Parallel Computers," *IEEE Transactions on Software Engineering*, vol. 26, no. 3, pp. 197–211, March 2000.
- [5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.
- [6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *OOP-SLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005, pp. 519–538.
- [7] J. D. McPherson, "Next-Generation Gap," *Nature*, vol. 6, no. 11s, 2009.
- [8] E. R. Mardis, "The Impact of Next-generation Sequencing Technology on Genetics," *Trends in Genetics*, vol. 24, no. 3, pp. 133–141, 2008.
- [9] M. S. Waterman and T. F. Smith, "Rapid Dynamic Programming Algorithms for RNA Secondary Structure," *Advances in Applied Mathematics*, vol. 7, no. 4, pp. 455–464, 1986.
- [10] D. S. Horner, G. Pavesi, T. Castrignan, P. D. D. Meo, S. Liuni, M. Sammeth, E. Picardi, and G. Pesole, "Bioinformatics Approaches for Genomics and Post Genomics Applications of Next-generation Sequencing," *Briefings on Bioinformatics*, vol. 11, no. 2, 2010.