

Cell ProRt *Cell Programmable Runtime* : A Programmable Adaptive Runtime System for Cell BE

Final Class Project for CS380P... Calvin Lin and Karthik Murthy

Apollo Ellis University of Texas at Austin
apichit@cs.utexas.edu

ABSTRACT

We present a programming model interface for the Cell BE aimed at being flexible and supporting general parallelism on that platform. Our interface is lightweight and adaptive, and a simple model for expressing task and data parallel programs. We provide a thread pool and task queue system as a basis parallel applications on Cell. In addition to this familiar model, we support several additional programmable and tunable features in the system. We consider this feature set to be novel in some aspects along with being sufficient, expressive, and powerful as a tool set for programming Cell. We implement a scheduler, work stealing, mutex, barrier, and most interestingly dynamic control of parallel granularity and an API in which to implement data structures allowing our runtime to take advantage of locality between tasks.

General Terms

Filter, Dice, Work-Steal, Parent-Child-Wait

Keywords

Task Parallel, Data Parallel, Programmable, Runtime, Granularity, Locality

1. INTRODUCTION

The Cell processor which was originally designed for the gaming console market had in the past garnered considerable attention from the parallel programming community [0]. However our experience with Cell and indeed a general consensus in the community has led this group to believe Cell to be a challenging platform to work with. This motivates the idea of providing a programming interface that would ease the programmers load in dealing with Cell. While indeed this is our goal it is worthy of mention that we adopt a "no free lunch" philosophy when approaching this problem. Surely no programming model can optimally schedule and execute parallel computations in a non-trivial manner. Indeed it has been shown that to provide a parallel

schedule for computations that is optimal is a hard problem, at least as hard as List Scheduling which is NP-Hard.

Thus we leave it up to the programmer to construct the parallel algorithm. However we also try to facilitate the ability to schedule computations with domain specific knowledge by providing a programmable scheduler. In addition what we provide is a model and within which to operate that abstracts away some features of the underlying system and provides a framework within which to operate. We attempt to manage parallel algorithms in a sensible way without impeding any application development. We also provide a programmable and tunable runtime, along with a library for explicitly interfacing with the runtime in an effective way.

2. DESIGN: A PROGRAMMABLE MODEL

The runtime system consists of a thread pool, task queue, scheduler, work stealer, mutual exclusion objects, a controller for multigrain parallelism, and a buffer API used for building data structures. All of these features are either fully programmable or tunable. The thread pool size is controlled by the user, while the task queue and scheduler work together with user code to reorder tasks according to desired scheduling constraints. The work stealing can be aggressive or lenient based on specifications of queue sizes and parameters of the runtime. Mutex functionality is supported from within the system by communication mechanisms between processes. Granularity support is also fully programmable. User functions split bulk jobs into multiple finer grain tasks to utilize the full parallelism in the system should it become necessary. The buffer API provides a level of abstraction above the DMA interface of the Cell processor. It also provides information to the run time about which tasks can run when and where to exhibit good locality.

2.1 Work Queue

The Cell ProRt system design is centered around a task queue consisting of three layers. This layered buffer contains references to objects derived from a central task type Work. The Work class contains information about the type of work along with data to assist with granularity control and possibly scheduling information. The class is virtual and derived classes actually populate the buffers.

At the first level tasks are posted into the Prefilter buffer. This buffer is used to store tasks for scheduling into the Reorder buffer discussed in the next paragraph. The Filter Thread acts on the Reorder buffer's current contents

and the Prefilter's new tasks using filtering procedures to merge them into the Reorder buffer. The filtering procedures are user coded functions adhering to a function prototype which takes as arguments the Reorder buffer and the Prefilter, and provides as output a new second layer buffer. The Prefilter and the Filter Thread are the primary scheduling mechanisms of the system as opposed to secondary work stealing functionality which may also affect scheduling. Note that, the filters are used in conjunction with the buffer API to improve locality within executions.

The Prefilter reorders the Reorder buffer or ROB. The ROB is the main and largest storage facility for tasks in the system. The Reorder buffer feeds into the Ready buffer which is filled on demand from the ROB by the threads in the thread pool. Hungry threads acquire a lock on the Reorder buffer and transfer jobs from it into the Ready buffer. The Ready buffer is the third layer buffer. The Ready buffer's usefulness comes from its contention reduction properties. By accessing the Ready buffer for extracting work, threads in the thread pool have lower contention for the task queue which is also being accessed by the Filter Thread.

2.2 Thread Pool

The thread pool is implemented on the SPEs. While the task queue is distributed across the SPEs and the PPE. Each thread in the pool runs small kernel which implements the API system calls for the tasks to utilize. The threads also manage their own local task buffers, an inbox and an outbox. These buffers contain space for many tasks further reducing the contention on the task queue.

Each thread in the pool occupies a single SPE, and is responsible for that SPE over the lifetime of an instance of the system. The thread will poll the Ready buffer and add any tasks found into its inbox. It will then begin to work on the tasks locally. The system calls in the task itself may call back into the thread kernel asking to post new data or possibly post new filtering procedures for the scheduler. When the inbox buffer runs low it will be filled before completing more work, also the outbox may be flushed at anytime by a user call to post or flush. It may be useful to now associate the reader with the formal API.

2.2.1 Thread Pool API

PostWork(Work* Job)

This method is available from within user code. It is used to issue a new task into the task queue for scheduling. The method takes the virtual type Work expecting a reference to a concrete implementation of the type. PostWork puts a task directly into the outbox, which may or may not be flushed to the Prefilter on call of this function. Flushing is governed by the capacity of the outbox which is static.

PostWork(List<Work*> JobList)

This method is used from user code to issue a new task list into the task queue for scheduling. It is a list form of the above method.

PostLocalWork(Work* Job)

This method is also available from within user code. When tasks are posted using this call, the task is posted locally to this calling threads inbox without any interaction with the centralize task queue.

Flush(bool Block=false)

Flush is a user accessible call into the API which immediately tries to flush the outbox out to the Prefilter buffer.

2.3 Mutex

A lock library for Cell has been implemented and locks can be instantiated and passed into tasks at creation. Along with this support a declarative mutex can be added to a task and the runtime will attempt to acquire that lock automatically before running the specified task.

2.4 Work Stealing

Work stealing occurs between SPE inboxes. When a thread finds that the Ready buffer is empty. Upon returning from the attempt on the ready buffer it will try to work steal from another threads inbox. Note that this brings significant challenges into the implementation of parent child wait in our system, but was shown to be an optimal solution in many cases. We rely on work stealing along with dicing discussed in the next section for load balancing within the system.

2.5 Barrier : Parent Child Wait

Not yet implemented in the system, parent child will allow a task to post a new task and wait for it to complete by context switching out. This functionality can be used to implement Barrier. If the first task in the system were to simply spawn all subsequent tasks it would act a barrier waiting for those tasks to finish. We believe this to be an intuitive barrier implementation. It will be added to the system soon.

2.6 Granularity Control : Dicing

The granularity control in our system is integrated into the runtime scheduler. At every pass of the Filter Thread over the Prefilter and the Reorder Buffer we may place a call to the function Dice on each task object. The decision to dice or not depends on the current state of the Reorder buffer in particular if it is running low on tasks dice may be utilized. With the users implementation of dice available the call may return new tasks with a finer granularity and may even change the algorithm used in tasks. This way of providing adaptivity is useful for load balance and good system utilization.

2.7 Buffer API

The buffer API is used to implement data structures which may be managed by the runtime to provide locality in scheduling. The buffer API should allow for nearly arbitrary data structures to be implemented efficiently. Graphs, trees, arrays, lists, matrices, and data streams were in mind during design. Each buffer object is templated with user specified structures, and can be linked to other buffer objects to create groups and hierarchies. Each task can then be associated with a buffer object and further associate the tasks that it posts with their own buffer objects. These objects can be checked for at runtime within the scheduler to assign tasks more locally to each other if buffers are shared. The runtime system keeps track of buffer locations and can

schedule jobs to locations where their associated buffers already exist in local memory. Thus locality scheduling can be achieved with careful implementation of data structures and algorithms using the buffer API. The buffer API as of yet has not been fully specified in the runtime.

3. IMPLEMENTATION AND WORK FLOW

The original implementation for Cell cited here [2] used a very slow and inefficient scheme for implementing a simple task queue. Jobs ran with no memory safety and it was impossible to make calls to malloc or do any other type of heap operation within a task. The API itself comes from Sean Keely's work on SolMT an efficient run time for fine grained parallelism. The implementation here extends both works.

From the user perspective the API usage includes writing an SPE task and compiling using a spu-gcc or spu-g++. The compiled task must be imbedded in the a PPE program. From that program it is registered with the system. This is the typical work flow for coding on Cell we have changed nothing there. The registration function takes an external handle to the SPE task and registers by type. The user may then implement a class extending Work and create instances of the class for entry into the task queue. After assembling at least one task and possibly a list of tasks for entry, WorkQueueInit is called.

WorkQueueInit Initializes the system variables and task buffers. It then makes a call to SpawnWorkers which initializes the thread pool utilizing libspe2 and pthreads to place the SPE kernel binary on the actual SPEs. The function will also instantiate locks and other synchronization objects used for maintenance of the system. The SPE threads will be launched with knowledge of the address of all locks and buffers in the PPE memory. The PPE constructs the Prefilter, ROB and the Ready buffer. It passes this information into the SPEs. WorkQueueInit is also responsible for spawning the Filter Thread which is implemented as a Pthread.

The user should then place at least one call to PostToROB. This posts tasks into the Prefilter. The Prefilter is then released to place the tasks into the Reorder Buffer. At this time on the SPE side the worker threads will notice that the Reorder Buffer has become populated during their polling. One of the threads will then fill the ready buffer by doing a DMA on both the ROB and the Ready buffer, then filling the latter with some number of elements from the former. After filling the Ready buffer, threads contend for the lock on that buffer in some order and proceed to fill their own inboxes with work. Each buffer including the inbox has its own lock and this lock must be acquired before access the buffer. At this point SPEs can begin to work and possibly repeatedly do so until their inbox empties and they must return to the Ready buffer. If the Ready buffer becomes empty after some time it will be filled again in the same fashion by the first available SPE.

3.1 Locks

The lock API makes heavy use of the intrinsics mfcgetllr and mfcputllc. The former is a way to move a cache line from the PPE to the SPE and create a reservation on that line. This is implemented with snooping caches on the SPEs.

The mfcputllc command will succeed in replacing the cache line on the PPE from the SPE only if the reservation has not been violated by another mfcgetllr. The spinlocks used for the all the task buffers are implemented with a do while loop over a call to mfcgetllr and the corresponding call to mfcputllc. The address of the cache line used for the decremter in this lock is distributed uniformly to the SPEs at startup mentioned above.

3.2 Lists

We have rewritten the entire API from [2] for better adherence to Sean Keely's API work. In order to adhere to the API more strictly the list structure from the stl library had to be duplicated for Cell. The fact that memory is not shared on this platform and DMA manages transfers to and from SPE local storage provided a significant challenge. Our list implementation uses contiguous buffers for shuffling data on and off of the SPEs. While the buffers are contiguous in memory extra data is kept around to preserve the linked list properties of the structure. A self-adjusting list may be the proper term for the implementation. The data is kept in a templated buffer B and the links in another buffer I. Both buffers must be modified to add, remove, insert, and delete from a structure List L. B and I internal to L are both shipped in and out during DMA of the structure. This List was used to implement the Reorder Buffer, the Prefilter, and the Ready Buffer. The inbox and outbox are also composed of this list structure. This simplified the port of the x86 windows version of the API.

3.3 Nitty Gritty Details

Note that the binary for the kernel is linked with a special flag to set the stack pointer below the loading address for the tasks. The loading address for the task is compiled in as well. The stack pointer on must be set to point to the user stack at task launch time and the pointer must be set back to the kernel stack in between tasks and on system calls. This allows for memory safety with the tasks. Tasks may use malloc or any other heap operation safely.

4. RELATED WORK

4.1 Sequoia

Sequoia [0] is a programming language and and runtime system currently targeted at the Cell BE and cluster computers. The Sequoia is not a model aimed directly at parallel programmability. It is mainly designed to take advantage of locality. In sequoia parallel algorithms are decomposed into tasks. The tasks are of two variants; inner and leaf. The inner nodes are only allowed to distribute blocks of data in the form of arrays to child processes. The compiler and runtime manage when and where a task runs and when the leaf variant of a task is invoked as opposed to the inner task. This Sequoia imposes a divide and conquer algorithmic preference on programmers arguing for the usefulness of cache oblivious algorithms. Through recursion and block array mapping primitives tasks are eventually decomposed into leafs which run when their data fits in the unit of memory closest to the processor i.e. cache or local store. There is no communication across tasks, and therefore limited forms of synchronization are available. They implement a blocked matrix multiplication system with recursion similar to a cache oblivious implementation of the algorithm.

4.2 MARS

MARS [6] or the Multi-core Acceleration Runtime System from Sony Corporation of America is a SPE centric runtime for the Cell BE. It is based on a bulk synchronous task parallel work queue system. Tasks or SPE programs are scheduled by the runtime based on statically assigned priorities. They provide lightweight context switching and a library of synchronization objects. When tasks block on synchronization they can be context switched. Context switch also occurs in the case of parent child wait, which is also supported in the system. The programming model is argued to incur minimal host processor (PPU) load, and minimize host and peripheral (SPE) communication. It is also claimed to simplify max SPE utilization. MARS can out perform libspe and libspe2 when number of SPEs is significantly less than the number of SPE programs trying to execute. Programming in MARS is very similar to programming Cell ProRt the main difference is the hence a large part of the contribution of this paper is the adaptive nature of our runtime. In MARS a task is specified once and the granularity of the parallelism available is thus fixed. If the task queue becomes relatively low on tasks the remaining work can exhibit poor load balance. Also there is no direct support for locality in MARS or anything other than priority scheduling.

4.3 MGPS

MGPS [7] is an MPI based runtime and scheduler that attempts to address both task and loop level parallelism on the Cell. We believe this to be a limiting model as levels of granularity in parallel programs are not always expressible in either of two forms. In MGPS for every MPI thread, annotations denote offload-able functions to the runtime. The functions may be run either on the PPE or offloaded and run on a SPE if available. Thus the PPE load can be significant. They implement a custom MPI scheduler which they claim beats the Linux Kernel by a factor of 2. Loop level parallelism is also achieved with annotations.

As long as there are tasks to be executed MGPS is a task scheduler. When an SPE become idle MGPS begins to offload loop iterations from SPEs to SPEs. Again programmers may or may not prefer to specify their parallelism only as tasks with parallel loops. Also for complicated and perhaps disjoint application writing all tasks together into in single program may impose limitations and programmability issues. For example if a programmer writes code for a physics simulation and a graphics rendering, the codes would have to be compiled together into a single binary from a single code base. In fact the programmer would have to implement possibly complex multiplexing of work within a single MPI program to ensure all tasks are run efficiently. In Cell ProRt tasks are specified in a explicitly disjoint manner. Many different types of codes can be in the job queue at the same time with out having any knowledge of or interaction with each other. We do however also allow for communication and synchronization between tasks. In fact synchronization in the MGPS system is left completely up to the MPI implementation of the code. In other words SPE offloaded tasks cannot synchronize at all.

4.4 CellSs

The CellSs [4] compiler and runtime also try to address the issue of programming Cell by annotation of a single source

code. The issues of MGPS are inherently shared by this model. We see no major difference from MGPS in the CellSs contribution, aside from CellSs adding a locality aware runtime scheduler, and lacking of granularity control of any kind. The addition the locality aware scheduler attempts to handle RaW dependencies via runtime checks on a dependency graph of tasks. The scheduler trivially try match resources with ready tasks. They do not attempt to provide any way to prevent repeated fetch in multiple tasks utilizing the same data. This is another of our major contributions in the this paper, our scheduler can schedule tasks using the the same data and prevent an re-fetch to local memory.

5. FUTURE WORK

The API is still under construction in many facets. Although we state what the system *does* in the design, not all features have been successfully implemented thus far. Work stealing is not instrumented yet. The call to PostToROB is currently the only way in which to get tasks into the system. The API for the user code has not been tested and only programs which simply run to completion have been used for testing. Plans for the the future include but are not limited to the following.

We plan to write a mandelbrot set renderer to compete with the MARS version of this type of renderer. We believe they may have tuned there renderer to a particular view or for some granularity, we plan to use dicing and work stealing to construct a faster run in our system due to better load balance. In plain we will render an incredibly large set, and a set from many different views in both systems, then determine the better runtime and record the results. We believe the MARS system will exhibit over head on large sets if they have tuned their renderer for smaller sets with finer granularity. Also we believe that if they have tuned for a particular view of the set all their tuning for load balance may invalidated on other views. We hope to show that our work will out do the MARS implementation these facets by being adaptive and load balanced.

We also hope to compete with the Matrix Matrix multiply algorithm used in the Sequoia paper. We will implement SUMMA on Cell show that our feature set induces this type of flexibility in algorithm choice. We will of course also be exhibiting our full feature set in the implementation to measure our wins there. Sequoia also has some results published on CellStream, a simple spec in [3] used to look at the ability to reach good performance peaks with different models on Cell. We will run this spec in our model and provide results.

Finally we will implement the Buffer API and show a locality result in an irregular algorithm, ray tracing. We will be timing ourselves against [5] a very fine tuned by hand implementation of the algorithm. If we can be competitive we believe this would be a good result.

6. CONCLUSION

We present Cell ProRt a flexible and general parallelism solution. We have discussed the implementation of the current work and the plans for the future of the this work. We hope to show that coding efficiently on the Cell does not have to be cumbersome and difficult. We consider several other models of programming and while this list is not exhaus-

tive we believe it to represent the Cell programming model community well. We do not simply try to compete with the latest an greatest but try to find a solution that exhibits novelty, efficiency, expressibility and programmability while remaining elegant and simple.

7. ACKNOWLEDGEMENTS

Sean Keely, Okan Arikan, Donald Fussell, Calvin Lin , John Bates (Sony) and MARS team (Sony), Bharadwaj Subramanian

8. REFERENCES

- [0] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. 2006. Sequoia: programming the memory hierarchy. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (Tampa, Florida, November 11 - 17, 2006). SC '06. ACM, New York, NY, 83. DOI= <http://doi.acm.org/10.1145/1188455.1188543>
- [1] Williams, S., Shalf, J., Oliner, L., Kamil, S., Husbands, P., and Yelick, K. 2006. The potential of the cell processor for scientific computing. In Proceedings of the 3rd Conference on Computing Frontiers (Ischia, Italy, May 03 - 05, 2006). CF '06. ACM, New York, NY, 9-20. DOI= <http://doi.acm.org/10.1145/1128022.1128027>
- [2] Apollo Ellis, Bharadwaj Subramanian 2009. Cell-MT : An efficient runtime for fine-grained parallelism. Published On the Web @ <http://userweb.cs.utexas.edu/api-chit/Documents/view.pdf>
- [3] Schneider, S., Yeom, J., Rose, B., Linford, J. C., Sandu, A., and Nikolopoulos, D. S. 2009. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. SIGPLAN Not. 44, 4 (Feb. 2009), 131-140. DOI= <http://doi.acm.org/10.1145/1594835.1504197>
- [4] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. 2006. CellSs: a programming model for the cell BE architecture. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (Tampa, Florida, November 11 - 17, 2006). SC '06. ACM, New York, NY, 86. DOI= <http://doi.acm.org/10.1145/1188455.1188546>
- [5] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich Ray Tracing on the CELL Processor Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pages 25-23
- [6] MARS Presentations, <ftp://ftp.infradead.org/pub/Sony-PS3/mars/presentations/>
- [7] Blagojevic, F., Nikolopoulos, D. S., Stamatakis, A., and Antonopoulos, C. D. 2007. Dynamic multigrain parallelization on the cell broadband engine. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Jose, California, USA, March 14 - 17, 2007). PPOPP '07. ACM, New York, NY, 90-100. DOI= <http://doi.acm.org/10.1145/1229428.1229445>