Experiences with Hybrid Clusters

Damir Jamsek
jamsek@us.ibm.com

Eric Van Hensbergen
bergevan@us.ibm.com

***Abstract*—The complexity of modern microprocessor design involving billions of transistors at increasingly denser scales creates many challenges particularly in the area of design reliability and predictable yields. Researchers at IBM's Austin Research Lab have increasingly depended on software based simulation of various aspects of the design and manufacturing process to help address these challenges. The computational complexity and sheer scale of these simulations have lead to the exploration of the application of high-performance hybrid computing clusters to accelerate the design process.**

Currently, the hybrid clusters in use are composed primarily of commodity workstations and servers incorporating commodity NVIDIA-based GPU graphics cards and TESLA GPU computational accelerators. We have also been experimenting with blade clusters composed of both general purpose servers and PowerXcell accelerators leveraging the computational throughput of the Cell processor.

In this paper we will detail our experiences with accelerating our workloads on these hybrid cluster platforms. We will discuss our initial approach of combining hybrid runtimes such as CUDA with MPI to address cluster computation. We will also describe a custom cluster hybrid infrastructure we are developing to deal with some of the perceived shortcomings of MPI and other traditional cluster tools when dealing with hybrid computing environments.

I. Motivation

The VLSI department in the IBM Austin Research Lab is developing techniques to enable custom microprocessors to be designed reliably and fabricated with predictable yields. This is becoming increasingly more difficult in future 22nm and beyond technologies. More then ever, reliance on software simulation of various aspects of design and manufacturing is becoming necessary.

The algorithms involved have various computational needs. Our recent work has focused on codes for: electrical simulation of small circuits, power grid analysis for whole chips, lithography simulation for physical design, and logic simulation at the RTL level.

- electrical simulation - Monte Carlo simulation of small circuits (10's - 100's of devices)
- power grid analysis - shape manipulation and sparse linear algebra
- lithography simulation - primarily image processing (fft, convolution, bit map manipulation)
- logic simulation - Monte Carlo boolean gate evaluation

There are differing parallelization strategies for these tasks. The Monte Carlo simulations are conceptually simple from a parallelization view. The difficulty is in redesigning the simulation so that it both "fits" on a compute node and that resources it needs are either shared or distributed efficiently.

For tasks with core algorithms that are liner algebra, fft, convolution or shape and image-based manipulations, the core computations must be efficiently mapped to a multicore architecture. In addition, the same issues of resource allocation and data sharing or distribution exists.

The simulations in question have considerable computational requirements that on typical engineering workstations or clusters of workstations take days if not weeks to complete. In an environment where design changes are ongoing these turnaround times are unacceptable. Our goal is to reduce these turn around times by two or three orders of magnitude to hours or minutes. This will require hundreds or thousands of compute elements effectively cooperating on the tasks.

The computational complexity of these simulations has caused us to explore using large multicore and hybrid systems. In particular, clusters of workstations with multicore accelerators as well as large scale computing using systems like IBM's Blue Gene. In both cases, issues of resource allocation and management become as important as the task of algorithm redesign and implementation.

In this paper we will discuss our experiences applying heterogenous hybrid clusters to these applications and describe the various runtime infrastructures we used to address the problems within a cluster environment. The next section gives a brief overview of hybrid systems and discusses several hybrid cluster configurations we have experience with. In Section 3 we'll detail one of the applications in question and describe our various experiments with deploying it on a cluster environment. Based on these experiences we began to develop our own cluster runtime tuned to the needs of heterogenous hybrid computing which we shall describe in Section 4. We'll conclude in Section 5 by discussing our status and touching on some some topics for future work.

II. Heterogeneous Hybrid Clusters

Hybrid systems incorporate heterogeneous cores, either with different processing characteristics or difference instruction set architectures. The Cell Processor [1], with its combination of a general purpose PowerPC cores (PPU) and eight synergistic processing elements (SPE) for vector operations is a classic example. In such systems, the various cores have coherent access to each other's memory, but memory transfers between PPU and SPE are handled explicitly. Such a configuration can be seen in the top node in Figure 1.

The Road Runner [2] tri-blades used a set of Cell processors connected over peripheral buses to a general purpose Opteron blade (bottom node in Figure 1). Road Runner poses many

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

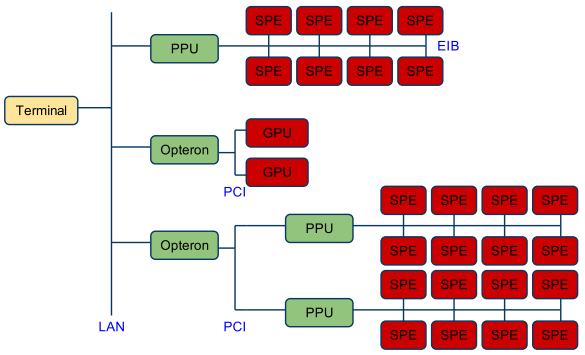

Fig. 1. Hybrid Topologies

interesting challenges as a hybrid as it has multiple instruction sets, multiple endian, and multiple peripheral buses.

Another tightly coupled hybrid can be seen in systems using GPUs to accelerate computation. In such systems, the GPU accelerators are often located on the other side of a peripheral bus with their own memory and execution units (middle node in Figure 1). A common complication is that in many cases GPU clusters have been built with a variety of off-the-shelf accelerators each with different memory configurations, different GPU models, and different numbers of GPU cores on every system. This creates a heterogenous environment which is difficult for traditional runtimes to cope with in a reasonable manner.

These sorts of hybrid nodes can themselves be composed together with other more general purpose systems into heterogeneous hybrid clusters. There has been a significant amount of work addressing the programming model and runtime of single node hybrid systems, but relatively little focus has been given to larger scale distributed contexts. IBM has a great deal of experience integrating heterogenous clusters and we are leveraging that history in our approach to hybrid computing.

III. Application Case Study

One area in which future chip fabrication will require significantly greater computation resources than in the past is lithography simulation. In its most basic form, lithography simulation is 2D image convolution done via FFTs as well as a few other image processing computations. GPUs are good FFT engines and imaging engines achieving between 50x and 100x speedup depending on the task on a 240 core processor NVIDIA G200 chip. The distribution of work to a cluster of GPUs is relatively straightforward as most processing is localized to a rectangular subblock of the the overall design with possible data shared between adjacent subblocks.

We initially developed a CUDA based lithography simulation on a single node with a G200 GPU. The simulation runtime was on the order of 1/10 sec for a 1 micron by 1 micron sublock compared to a runtime of 2-3 seconds on a conventional workstation. A modest size 100 mm^2 chip has 10^8 of these sublocks. Taking advantage of symmetry in the design we would hope to reduce this by a couple orders of magnitude leaving 10^6 subblocks to be imaged. At 10 per second that leaves 10^5 seconds, or just over 27 hours (as opposed to 22 to 33 days on a conventional system). Imaging a more complicated chip in the range of 600 mm^2 would take more than a week of computation (compared to 5-7 months on a conventional system). A cluster of hybrid machines would bring this back to a sub-day simulation time necessary for quick iteration of the design process.

Our initial cluster implementation for lithography simulation used a simple MPI configuration with statically defined workload distribution based on one MPI task per available node. The cluster nodes in question were commodity Opteron and Woodcrest based desktop workstations with 4-16GB of memory and one or two NVIDIA based high-end graphics cards (some of which had 2 GPUs on the card). The back-end nodes ran a common version of a 64-bit RedHat Enterprise Linux Client version 5.2 and CUDA 2.2. To complicate things slightly we ran our master node on a Intel-based MacBook Pro using its internal GPU to visualize the simulation.

The master simulation thread defines the task, preconditions the design data and distributes it to nodes on the cluster using MPI Send. Each GPU task has an entire copy of the design data or possibly some subset and waits for MPI distributed instructions to perform some subset of the computation. Depending on the task requested the GPU tasks may hold the data or return some subset using MPI. In general, for large designs, the data computed may be larger than the ability of any one MPI node to hold and must remain distributed or returned to some shared storage area. Cluster based runs scaled linearly due to the nature of the workload and simulation runtime reduction matched our initial expectations.

While the MPI based runtime achieved the desired results, it presented several barriers to production deployment for our circuit designers. Our initial implementation used a dedicated set of resources, but for production we needed a cloud-like configuration where multiple designers could attach to accelerator resources without conflicting with each other in a dynamic fashion. Since our intended configuration involved nodes with several GPUs we didn't want to dedicate resources at a node level, but rather at a GPU level. Even though we had a relatively small cluster and a naive MPI configuration, there was a tremendous degree of difficulty in setting up and maintaining the MPI hosts. Additionally, we found ourselves using relatively little of the features of MPI, relying on it for work distribution and rudimentary data communication that could just as easily been done directly with files or even over standard I/O. The worst aspect of the MPI runtime was that it had no direct knowledge of the underlying configuration of the hybrid accelerators requiring us to distribute work based on the lowest common denominator of GPU capabilities. This lead to many GPUs (particularly on nodes with 2 or more GPUs available) to be underutilized. The complexity of MPI combined with its lack of any knowledge of GPUs lead us to look for other distributed execution models.

IV. Hybrid Workload Management

After a brief search of cluster management software yielded no solution which incorporated explicit support for hybrid accelerators such as GPUs we developed a new hybrid cluster infrastructure based on our unified execution model [3] named KIRIN after a mythological hybrid creature of the far east.

A. Unified Execution Model

The Unified Execution Model (UEM) provides an extensible and flexible workload management middleware which end users and applications interact with directly through a synthetic file system much like the proc file system pioneered by UNIX and later extended by Plan 9 and adopted by Linux. Within these systems, every process on the system is represented by a file (in the case of historical UNIX), or a directory (in the case of Plan 9 and Linux). In the latter case, there are a number of synthetic files within each process' directory providing information, events, and control interfaces.

The UEM takes the control of processes via a synthetic file system one step further, enabling process creation, control, and inter-application pipelining (in the spirit of Unix pipes), across multiple nodes. Importantly, this interface is *distributed*, eliding the need for a central control or coordination point, and facilitating scalability. End-user workstations can also participate directly in the unified execution model, allowing local scripts and management applications to interact directly with the cloud distributed infrastructure when desired. The UEM utilizes Zeroconf based registries and integrates its own authentication mechanisms minimizing the need for per-node configuration.

B. KIRIN Extensions

For hybrid systems we extended the basic model via its plug-in interface to allow resources to publish the existence of accelerators (such as GPUs) and their attributes (such as the number of cores, the available GPU memory, bandwidth, and so forth). We then extended the provisioning interface to allow new job requests to specify preferred and mandatory attributes for the nodes the job would be executed on.

The UEM approach is to use a task-based organization – the initiating thread establishes a cross-node session incorporating all the node-specific thread components which make up the execution of a particular task. The mechanism behind creation of the session as well as initiating execution of the particular node threads is based heavily on XCPU's example [4] of using a special file, conventionally named *clone*, to allocate new resources. Clone files have been commonly used in Plan 9 synthetic file servers to atomically (from a file system perspective anyways) allocate and access an underlying resource. This clone file creates a new task session and creates a synthetic subdirectory labeled with a unique id. Subsequent reads and writes to the file descriptor which was opened as the clone file will be directed to the control file of that session subdirectory following the typical clone semantic of Plan 9.

The first commands sent to this control file specify the desired resources for task execution. The UEM already allows us to specify characteristics of the host system (ISA, memory, disk, network bandwidth), and KIRIN allows us to specify additional attributes (GPU Type, Number of Cores, GPU Memory, Bandwidth, etc.). Parameters on the attributes allow specification of desired attributes versus required attributes. The application may specify whether it wishes to block waiting for the requested number of resources, or may get feedback from the infrastructure on the currently available resources.

From an implementation perspective, we really have two different organizational elements: physical resource allocation and task based execution. Since hybrid models such as CUDA operate as black boxes, we must be able to support dedicating hardware to a particular task as well as standard time sharing models. Once the specific node based resources are allocated, they materialize as uniquely identified subdirectories within the session directory. They may then be interacted with as a group through the control files of the session directory, or individually through control files in each of the subdirectories. The application can query the infrastructure for the particular attributes of the resources it was able to reserve. It may then modify its partitioning of the workload (or even the algorithm) to match the available resources.

In a fashion similar to Plan 9's *cpu* command, KIRIN will also coordinate access to the control node's local resources (such as the file system which the executable resides in) and make those available on the target node in the name space the command will be executed in. Since KIRIN operates within a private name space on the back-end nodes and imports its file system from the front-end, each task operates within its own container. With this methodology you can execute multiple isolated tasks on the same node with completely different Linux distributions or versions. Following the convention of Inferno's devcmd [5] and XCPU we initiate execution by writing a command to the open control file handle detailing the (local) path to the executable and command line arguments. Other configurations, such as alternate name space configuration, environment variables, etc. can be specified either through direct interaction with the control file or through other file system interfaces. The remote node will then setup the name space and environment and initiate execution of the application, redirecting standard I/O to the control node.

C. Example

Our approach is perhaps best illustrated by walking through an example. What follows is how one might use KIRIN to implement the lithography application described in the previous section.

The master thread parses the image file to a sufficient degree in order to determine an upper bound on the number of logical instances which would be required to process the image completely in parallel. The user, either by parameter, configuration file, or environment may limit the maximum number of requested logical GPU instances.

For the purposes of this example, lets say the image has 100 tiles to process. The master thread issues a command to reserve 100 GPUs, and gets an error response that only 8 GPUs are

available. The master thread recomputes an appropriate split of the work and re-issues the request asking for 8 GPUs, which succeeds. This creates 8 new subdirectories in the task session directory, each representing a thread on the local and/or remote system which has been allocated as a GPU resource. Since it is executing the same subject thread on the resources it can use the existing file descriptor to issue the execute command to all subject threads. It then iterates over the thread subdirectories, querying the particular capabilities (number of cores, memory, etc.) of each GPU and adjusts the workload partitioning of each thread to fit the capabilities of the hardware.

Fig. 2. KIRIN Task FSAPI

The master thread, who has references to the standard input, output, and error of each subject thread instance issues work assignments by sending a relative path to the data file along with coordinates specifying the tile that the thread in question is instructed to process. As threads complete their assigned work, they can either write an image file of their own to the shared file system or send the results over standard output directly to the master thread. Upon receiving results, the master thread can issue more work, or instruct the subject thread to terminate and release the resource.

When finished, the master thread can cleanup the allocated resources by closing the handle to the control file. Individual threads can be shutdown or retasked by opening their individual control files present in the subject thread subdirectories. Additionally, new resources can be requested and new threads can be started at any point during execution to explore different design areas in more detail.

V. Discussion

We are currently actively developing this hybrid-enabled unified execution model and testing it on our prototype cluster. It is our intention to develop this infrastructure into a common framework that can be used by applications to leverage resources from many different cluster configurations in order to maximize efficiency and performance. The end result should be a framework which allows us to setup workload pipelines between conventional systems, hybrid systems, mainframes and extreme scale systems such as Blue Gene. Resources from each of the clusters will be acquired and released dynamically and interconnected in a language and system independent fashion.

Existing runtime frameworks for GPUs, such as CUDA [6], provide precious little introspection and control to the external system – making monitoring and control difficult. Hopefully NVIDIA and other GPU vendors will open more of their interfaces allowing finer granularity of control and proper time sharing and virtualization of GPU resources. We intend to explore more granular facilities using the Cell hybrid processor which has support for both time sharing and virtualization.

Our existing prototypes all interact directly with the synthetic file system, but we are working on providing a simple library API with multiple language bindings. Specialized versions of this library can be constructed for certain classes of application and provide more advanced facilities such as Cilk-like work stealing, data pipelining between computational elements (ala PUSH [7]), or transactional semantics enabling either checkpoint/restart or triple modular redundancy of computation.

While we currently support adding and removing resources while running a task, we'd like to extend this functionality to deal both with failure and grid-like environment where idle compute resources may be donated by end users. In such an environment, we'd like to establish callback paths to the application so that it can leverage feedback optimization to adjust its partitioning and algorithms as resources change dynamically underneath it. Eventually, it would be nice to incorporate these facilities into runtimes which manage issues involving different instruction set architectures or facilitate just-in-time compilation from a universal byte code such as LLVM [8] to the target platform.

This work has been supported by the Department of Energy Of Office of Science Operating and Runtime Systems for Extreme Scale Scientific Computation project under contract #DE-FG02-08ER25851.

References

[1] IBM, "Cell broadband engine architecture, version 1.0," 2005.
[2] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, 2008.
[3] N. Evans, E. Van Hensbergen, and P. Stanley-Marbell, "Unified execution model," *Proceedings of the SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.
[4] L. Ionkov and E. V. Hensbergen, "XCPU2: Distributed seamless desktop extension," *International Conference on Cluster Computing*, 2009.
[5] "Inferno Man Pages," *Inferno 3rd Edition Programmers Manual*, vol. 2.
[6] J. Nickolls and B. I, "NVIDIA CUDA software and GPU parallel computing architecture," *Microprocessor Forum*, 2007.
[7] N. Evans and E. Van Hensbergen, "Push: a DISC shell," *Proceedings of the Principles of Distributed Computing Conference*, 2009.
[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *CGO '04: Proceedings of the international symposium on Code generation and optimization*, p. 75, 2004.