SLOWER: A performance model for Exascale computing

Thomas Sterling[1]***, Daniel Kogler***[1]***, Matthew Anderson***[1]***, Maciej Brodowicz***[1]

A performance framework is introduced to facilitate the development and optimization of extreme-scale abstract execution models and the future systems derived from them. SLOWER defines a six-dimensional design trade-off space based on sources of performance degradation that are invariant across system classes. Exemplar previous generation execution models (e.g., vector) are examined in terms of the SLOWER parameters to illustrate their alternative responses to changing enabling technologies. New technology trends leading to nano-scale and the end of Moore's Law demand future innovations to address these same performance factors. An experimental execution model, ParalleX, is described to postulate one possible advanced abstraction upon which to base next generation hardware and software systems. A detailed examination is presented of how this class of dynamic adaptive execution model addresses SLOWER for advances in efficiency and scalability. To represent the SLOWER trade-off space, a queue model has been developed and is described. A set of simulation experiments spanning ranges of key parameters is presented to expose some initial properties of the SLOWER framework.

1. Introduction

As technology advances, the means to effectively benefit from its improved properties changes, sometimes significantly, in computer architecture, programming models, supporting system software, and other aspects. Together these reflect a crosscutting execution model, which must evolve to respond to opportunities and challenges of the emerging enabling technologies. But the choice space is often large at least in detail as execution model alternatives and system designs are explored. Guidance is required to govern derivation of new execution models and the design of underlying systems incorporating their innovative concepts. Currently, the challenge of realizing extreme-scale operation is one of efficiency and scalability within the constraints of energy and reliability.

The history of high performance computing is punctuated with dramatic paradigm shifts resulting in a succession of innovative execution models as the device technologies have progressed. But they share in common a set of key factors to which each has responded. Together these factors constitute a performance framework, within which past and future execution models can be analyzed and evaluated as well as compared. Such a performance framework can serve as the needed guidance of future system development by characterizing the trade-offs in structure, semantics, control, and enabling mechanisms. This paper describes one such performance model, SLOWER, that is serving in this capacity for research into extreme-scale execution models, runtime system software, and programming interfaces.

The breakdown of Dennard scaling [8] and the nearing end of Moore's Law in conjunction with the dominant limitation of power consumption are posing the greatest challenge to continued performance growth in two decades. Radical departures from previous conventional massively parallel processors (MPPs) and commodity clusters now incorporating multicore sockets and graphics processing unit (GPU) accelerators are requiring innovations in system design, operation, and usage. To optimize these new class of systems requires a performance framework to guide design evolution in a meaningful and quantifiable trade-off space.

The experimental SLOWER performance framework establishes a six-dimensional space of consideration; each letter of the acronym reflecting one of the dimensions. Starvation reflects

[1] Center for Research in Extreme Scale Technologies, School of Informatics and Computing, Indiana University, Bloomington, IN 47408, USA

an insufficiency of concurrent work to keep all the critical resources engaged. Latency quantifies the response time distance (usually in a measure of time, e.g., cycles) for remote access and service requests. Overhead distinguishes the work needed to manage parallel resources and task scheduling but that does not actually contribute to the useful computational work itself. Delay due to waiting time incurred due to resource access conflicts captures the last of the key dependencies of performance degradation. But two remaining factors contribute indirectly to effective system usage. Energy and its rate of consumption (power) is a key parameter in the raw sequential performance potentially affecting both logic voltage and clock rate. The final parameter, reliability, directly impacts availability of the system and therefore the percentage of time the system can be employed for useful work. Together these can expose the dominant contributors to the effects related to performance degradation.

Unfortunately, at this time, there is no adequate formalization of the SLOWER framework; no unified set of equations that fully captures the subtle interrelationships among the contributing factors. To make the situation worse, these parameters are not purely orthogonal; therefore a perturbation of one may directly alter another. This makes the framework less than ideal for direct analysis and application to distinguish among possible future systems and programming models. Recent work on exploring the SLOWER trade-off space is presented and discussed in this paper. SLOWER is described in detail with quantifiable units of measure. A table is provided that categorizes the history of multiple execution models over the last forty years in terms of their response to the SLOWER factors and their respective technology base. An example of a possible future class of execution models is represented by the experimental ParalleX model. ParalleX is described in some detail and then how it addresses the SLOWER factors through the assumption of dynamic adaptive techniques is discussed to show that alternative paradigms may be possible to enable future extreme-scale computing. To move this exploration from the qualitative to quantitative, a set of experiments are being conducted through the use of queuing simulation for parameter sweeps to expose the resulting trade-off space. Early experimental results are presented that exhibit partial sensitivities among the dominant parameters and inform consideration of design alternatives. The paper closes with summary conclusions and immediate future work.

2. Related Work

The most important historical performance model is Amdahl's law [2] which provides a simple expression for the maximum expected speedup of an algorithm using multiple processing units while disregarding issues of latency, overhead, and contention. Among performance models which do consider the impact of latency and overhead, the LogP [6] and related family of LogGP [1] and LogGPS [10] models have long formed the core of parallel computation modeling. LogP, summarized as **L**atency, **o**verhead, **g**ap required between two send/receive operations, and **P**, the number of processors, enables performance modeling with distributed memory and communication between processing units. Subsequent extensions of LogP incorporated longer messages (LogGP) and synchronization costs (LogGPS). These models have been extremely successful in guiding algorithm development and performance using conventional programming model approaches such as the Message Passing Interface (MPI). Before LogP, the Parallel Random Access Machine, or PRAM, model [9] also functioned as a performance model to guide parallel algorithm development by examining performance for multiple processors having constant time access to a shared memory. While a simple model, the flat cost of PRAM did not

closely model real behavior for the multiple layers of communication systems in modern architectures. Vector Random Access Machine [3] (V-RAM), a vector performance model incorporates instructions for vector operations in the context of sequential random-access memory.

The LogP family of performance models have been used to analyze performance of many MPI based applications. However, the rise of multi-core architectures has resulted in new performance models that recognize multi-threading. The Multi-core LogP (MLogP) model [5] extends LogP while redefining the P parameter in terms of the amount of computing power available rather than just the number of processors in order to better model performance of many-tasking approaches. The $\text{Log}_\text{N}\text{P}$ [4] extends LogP to incorporate the impact of memory and middleware with the capability to model point-to-point communication in clusters of Symmetric Multi-Processor nodes (SMPs). SLOWER, in contrast, provides a non-LogP based framework for multi-core architectures with a focus on medium and fine-grained parallelism while also incorporating the Exascale relevant issues of energy and resilience.

3. SLOWER

A performance model can serve as a means of evaluating the effectiveness of execution models, distinguishing among distinct execution models, and devising improved execution models with respect to available enabling technologies. A useful performance model is capable of describing sensitivities of performance metrics to contributing factors. The SLOWER performance model is introduced to provide a framework for evaluation and analysis of execution models. It guides their development and the implementation of advanced systems the structure and operation of which they govern. SLOWER is derived from a set of key parameters that contribute to the degradation of performance. The units of each parameter translate to metrics of time. Throughout the history of high performance computing these factors have been addressed directly or implicitly in the context of prevailing underlying technologies through the design of architecture, methods of parallel programming, and the mediation and control of system software. These foundational parameters have been invariants across the many generations of supercomputers and are a framework with which to track the transitions in computing systems and methods.

A formulation gives overall average performance in terms of these parameters is given in Eqn 1:

$$P = e(L, O, W) \times s(S) \times \mu(E) \times a(R) \tag{1}$$

where P is average performance, e is efficiency ($0 < e < 1$), L is latency, O is overhead, W are delays resulting from contention for shared objects, s is scalability or concurrency, S is starvation, μ is the normalized per execution unit per cycle performance, E is the power factor (energy) that determines clock rate, a is the availability of resources ($0 < a < 1$), R is the reliability that determines down-time events.

Equation 1 identifies the critical SLOWER parameters and shows their interrelationships and contributions to the overall performance. Both the efficiency and availability are fractional. The availability measure takes into account those operational issues that can detract from a system's ability to perform real work. One of these, R, is the Mean Time Between Failures (MTBF) of the system. This combined with scheduled downtime and maintenance time (time to recovery) determines the fraction of the total time a system can actually deliver user computation. This

parameter can be extended to incorporate scheduling conflicts with other workload in the job stream.

There is an assumed peak single instruction stream rate of operation, μ, that combines the register-to-register operation rate with the average load/store memory access time, which is admittedly application and cache hierarchy dependent. Here, however, is exposed the relationship between the clock rate and the power consumption, E. Within a restricted range, clock rate and therefore μ is proportional to the rate of energy consumption (power).

Scalability, s, is the amount of concurrency or number of execution units (e.g., cores) that are allocated to a computation at the same time. Starvation, S, is the inverse reflecting the absence of work to be performed through concurrency. Starvation is either due to an insufficiency of pending or available work to be performed or an imbalance of the work distribution across computing resources (some have too much while others too little).

The efficiency, e, of usage of the system is the ratio of the sustained rate of operation execution to the peak rate. As shown in this relationship a number of factors contribute to the efficiency. Latency, L, is the time to make a remote access or service request (not including the overhead of doing so) of an unloaded system. Latency can be measured in clock cycles. The delay, W (waiting for delayed access) incurred for access contention to shared logical or physical resources takes into consideration the effects due to a loaded system. Such access conflicts can occur for networks, memory banks, or synchronization objects (e.g., barriers). Overhead, O, is the additional work required to manage parallel resources and task scheduling beyond the actual useful work of the computation itself. Overhead consumes resources (and cycles) that could otherwise be employed for useful work thus reducing the relative efficiency of system operation. From these parameters is derived the SLOWER performance model framework.

While the performance relationship is valid for any particular operating point on average, it is imperfect in deriving a deeper understanding through sensitivity analysis because the dominant parameters are not orthogonal. One trivial case is that parameters measured in cycles may vary depending on the energy consumption rate, E, that causes the clock rate to change and therefore the cycle time. More significant is that the efficiency, e, and the scalability, S, are interrelated. Usually over part of the range of system scale, as S increases, generally e decreases. More subtly, overhead not only wastes cycles, it imposes a lower bound on the effective granularity that can be employed. For fixed size work (strong scaled), this puts an upper bound on parallelism and therefore an asymptotic limit on delivered performance. In spite of these shortcomings, SLOWER allows developers of parallel execution models and system (hardware and software) architectures to reason formally about design choices, sensitivities, and projected capabilities.

SLOWER does not directly reflect one other important quality metric, productivity. This parameter is poorly understood and not formally defined. Nonetheless, notionally it not only embodies the issues associated with SLOWER, it also combines the other life cycle issues such as programmability, generality, performance portability, and their trade-offs with delivered performance. Productivity is outside the scope of this work but crucial to establishing long term strategies and direction of future extreme-scale computing systems and methods.

4. Experimental Setup

To demonstrate the viability of the SLOWER execution model, a simulation tool called "JaamSim" [11], developed by Ausenco, was used to simulate abstract machines with a variety of parameters. The parameters considered included network latency, instruction mix, overhead

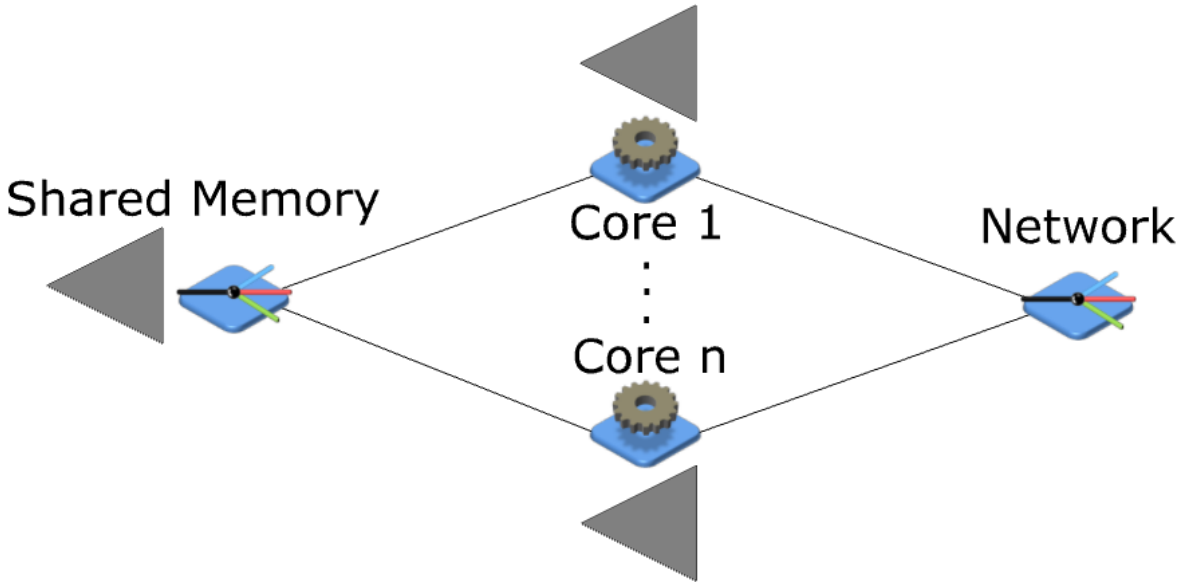

Figure 1. This JaamSim model builder diagram expresses the general flow of work in simulations. There are three types of components present: cores, the network, and memory. Each gear-component represents a core, and the cores are labeled from 1 to n for some n declared in the simulation parameters. The components with a three-way branch symbol (i.e. the network and memory components) indicate that any task arriving at that component is sent back to its place of origin after the branch component's processing is completed. Each component with a triangle next to it indicates that the component makes use of a queue data structure for handling work (in this case, lightweight tasks). Only one task may be worked on per core at a time. Likewise, only one memory request can be serviced by the shared memory at a time. Additionally, if a core sends a task to the memory unit, that core halts processing of any further tasks in its queue until the task returns from memory.

for context switching and network interaction, available parallelism per core, and number of cores per node. Here, available parallelism per core is represented by the number of lightweight tasks ready to run waiting on that core. Effectively, starvation, latency, overhead, and some effects of contention are modeled in these simulations, while energy and reliability are left out.

Each simulation assumed a single shared memory unit across all cores, and unlimited network bandwidth. Additionally, the lightweight tasks are assumed to never complete. In other words, available parallelism is held constant throughout each simulation and context switching overhead is only encountered when tasks switch due to a network operation.

For all simulations, the performance gain of programs using non-blocking protocols for network operations over programs using blocking protocols is measured. To compare the relative performance, each set of simulations was run twice. The first time using the parameters as described above with a non-blocking protocol, and the second time using identical parameters (except for overhead) while using a blocking protocol. The simulations of programs using a blocking protocol were all given an overhead of zero for context switching on network operations, as the program waits for the network operation to complete rather than changing tasks. An example simulation diagram can be seen in Figure 1.

In the context of this work, programs following blocking protocols temporarily cease performing useful work whenever a network operation is encountered. Work is resumed upon completion of the network operation. Note that only the core on which the network operation is invoked

stalls during the network operation. No context-switching occurs for these programs except upon task completion (which these simulations assume does not happen).

Non-blocking programs also temporarily cease performing useful work upon encountering a network operation, but do not wait until the network operation completes before resuming work. Rather, once the network operation has been invoked, the program performs a context-switch if there are other tasks in the program able to work. If no tasks are ready, then the program switches to the task that will be ready the soonest and then stalls until that task is ready to proceed with useful computations. The program will always perform a context-switch upon encountering a network operation, even if the context-switch overhead is greater than the network latency. This is because the program is assumed to not have knowledge of the time required to perform a context-switch nor of the network latency.

Performances of different simulation setups were compared via the total number of operations each setup completed in a given constant amount of time. This constant time was set to be the time required for 2^{17} register operations (reg-ops) to complete in the simulation. All durations in the simulation are normalized based on the time required for a register operation to complete. The choice for using 2^{17} reg-ops was made because this number proved sufficiently large to hide any significant noise in the performance results of simulations with network latencies of 8192 reg-ops (the largest latency simulated).

In all simulation models, the access time for local memory (excluding effects of contention) was set to 200 reg-ops. However, not every memory operation access the shared local memory. The simulations also take into account the effects of cache performance. The cache performance of all simulations is defined such that the programs run with a 90% L1-cache hit rate and a 90% L2-cache hit rate, and the hardware has L1-cache access times of 1 reg-op and L2-cache access times of 10 reg-ops. Additionally, the instruction mixes for all simulation models always include exactly 68% register instructions (excluding network and context-switching overheads). The remaining instructions are divided between memory accesses and network operations.

Four sets of simulations were run. The first set examined the effects of varying the overheads on the overall performance of the system. For these simulations, the model applied a constant overhead for each network access that represented the cost of context switching and network communication. The overheads applied ranged from 1 to 8192 reg-ops. The number of cores used in the simulation ranged from 1 to 32. All other parameters were kept constant. The network latency was set to 8192 reg-ops; each core was provided 64 tasks; and the instruction mix contained 8% network operations and 24% memory operations.

The second set of simulations explored the effects of having differing numbers of lightweight tasks available per core for various numbers of cores per node. The purpose of these simulations was to visualize the relative effects of available parallelism versus memory contention. For each of these simulations, the network latency was held constant at the equivalent of 8192 reg-ops; context-switching overhead was set to 16 reg-ops; and the instruction mix contained 8% network operations and 24% memory operations.

The third set of simulations examined the effects of changing the amount of contention experienced by an application (albeit limited to memory contention) with respect to various network latencies. This was accomplished by varying the number of cores present in the system. For these simulations, the number of cores was varied from 1 to 32. Network latency was ranged between 64 and 8192 reg-ops. All other parameters were kept constant. Overhead was set to 16

reg-ops; each core was given 64 lightweight tasks; and the instruction mix contained 8% network operations and 24% memory operations.

The final set of simulations examined in more detail the interrelationship of latency and starvation given a variety of instruction mixes. Because one form of starvation is the lack of parallelism (the other form being poor load-balancing), starvation's effects are examined by varying the number of available lightweight tasks per core. The network latency ranged from only 64 reg-ops to 8192 reg-ops, the number of available tasks ranged from 1 task per core to 128 tasks per core, and the instruction mixes contained anywhere from 0% to 16% network operations. Context switching overheads were kept constant at 16 reg-ops.

5. Experimental Results

The first set of simulations looked into the effects of overhead on the performance of an abstract machine running a program with a particular instruction mix. The results of this set of simulations can be seen in Figure 2. The graph presented shows the performance gain of a program using a non-blocking protocol when performing network operations as compared to a program that blocks on network operations.

As the graph indicates, there is a clear plateau in the performance gain of non-blocking programs over blocking programs. This plateau occurs when two conditions are met. The first condition is with regards to the effect of the overhead of context-switching and the number of available tasks waiting on each core. When the number of tasks multiplied by the overhead of a single context switch is less than the network latency, then we can achieve little further performance gain by continuing to decrease the overhead. This is because at this point, further decreasing the overhead increases the probability that all tasks on that core will end up waiting on network operations. The second condition relates to memory contention, which increases as the number of cores sharing the memory increases.

In order to further explore the plateau phenomenon, the second set of simulations was run. The difference in this set of simulations is that the tasks per core is varied and the overhead is held constant, whereas the previous set held a constant availability of tasks but with different levels of overhead. What is gathered from these experiments is that both decreasing the overhead and increasing the available parallelism lead to an exponential increase in the performance gain, up until the point where the product of the two equals the network latency. At this point, a plateau is reached. What is interesting is that in the former case the product is decreasing yet in the latter case the product is increasing.

This may seem contradictory to the previous results at first. After all, the previously mentioned effect reached its plateau when the latency was greater than the product of overhead and number of tasks. Closer examination reveals that these are in fact different scenarios. In Figure 2, the limiting factor in the performance gain was the lack of available parallelism, i.e. starvation. Decreasing overhead would not negate the effects of starvation. In Figure 3, the limiting factor is this time the overhead of context-switching. Adding more tasks to the system will not decrease the limiting effects of the overhead.

The results of the third set of simulations can be seen in Figure 4. The same plateau as in Figure 2 is visible here as well, though here the effect can be seen as limited to the higher latencies. This is because for lower latencies it is no longer possible for all tasks of a single core to be waiting on the network simultaneously. However, this does not explain the sudden decrease in performance gain that occurs as the number of cores is increased. The expected trend

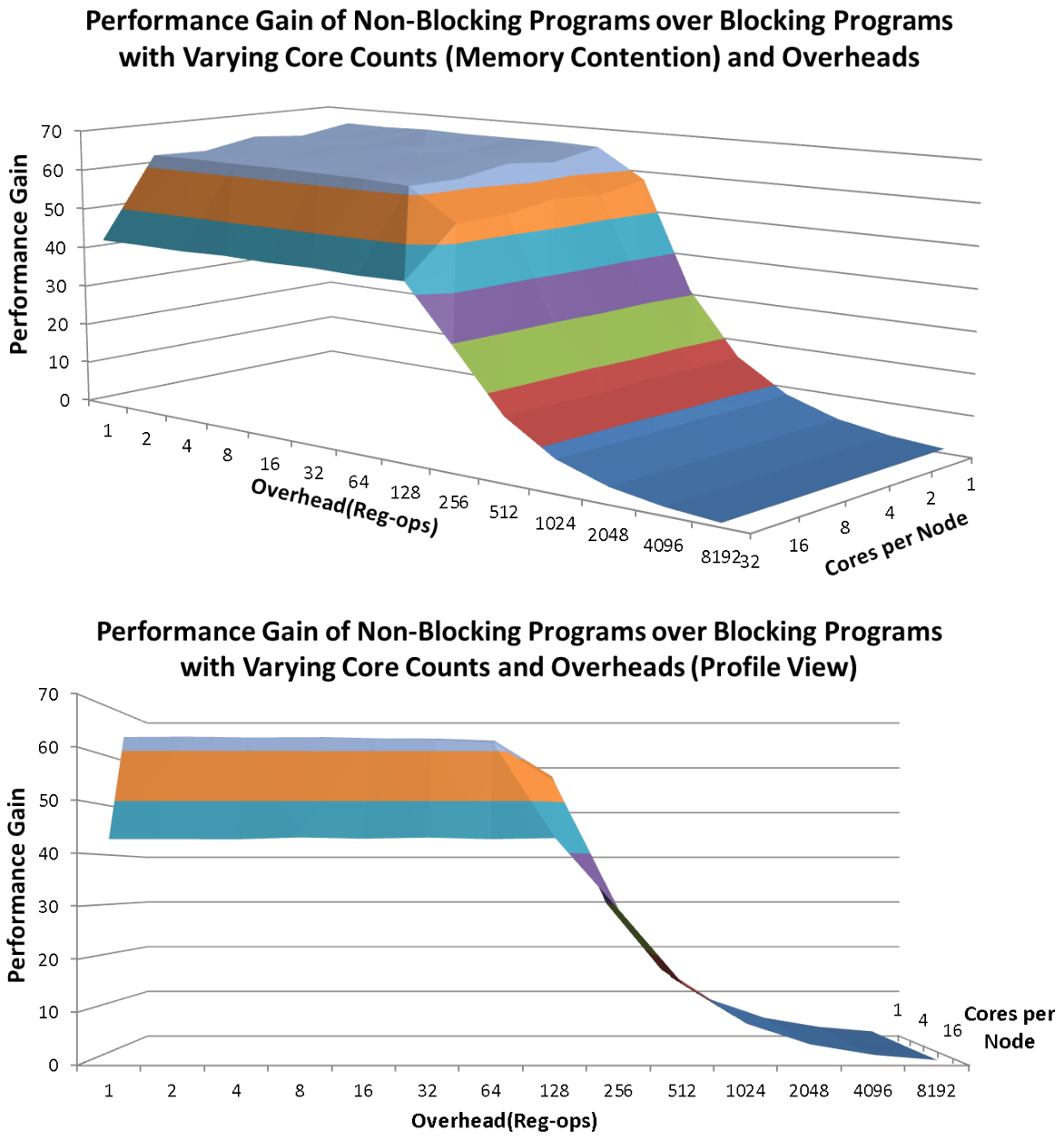

Figure 2. The performance gain of programs that do not block on network operations over those that do. This plot assumes a network latency equivalent to 8192 register operations. The number of parallel tasks available on each core is 64. The instruction mix consists of 68% register operations, 24% memory operations and 8% network operations. Overhead is measured with respect to the number of register operations that could complete in the same elapsed time. This figure demonstrates the deterioration in performance gain as both context-switch overheads increase and the number of cores increases. The latter decrease is due to increased contention for memory resources experienced by the non-blocking programs. The plateau in performance gain begins to occur where the product of the overhead and the number of tasks available equals the network latency ($128 \times 64 = 8192$). At this point the available parallelism begins to be exhausted.

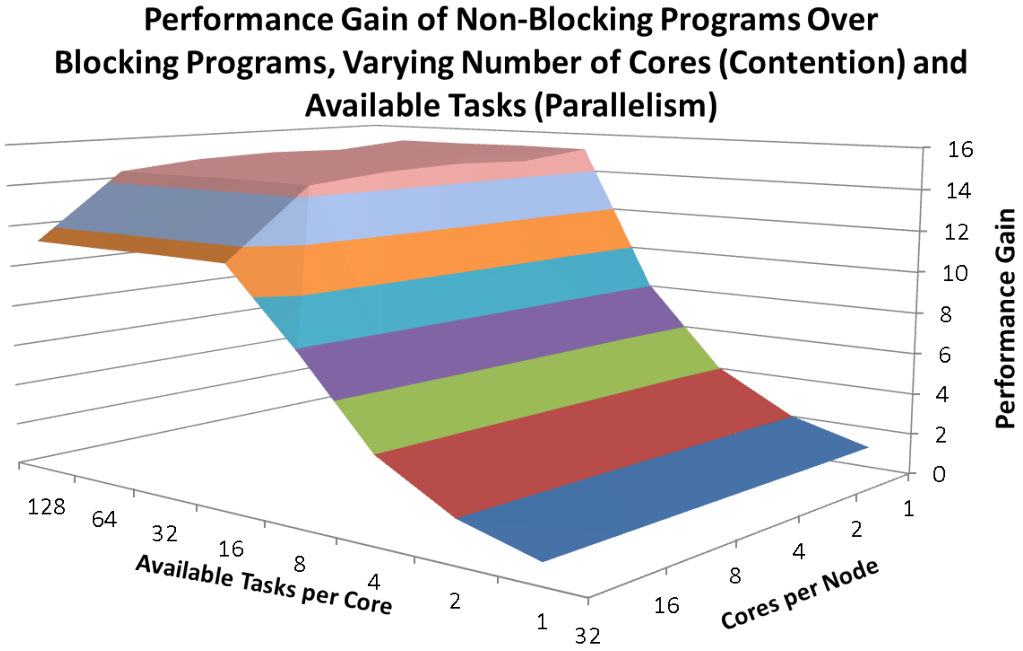

Figure 3. The performance gain of several simulations using non-blocking computations as compared to identical simulations that block computations upon accessing the network. All simulations were given an instruction mix consisting of 68% register operations, 8% network operations, and 24% memory operations. The network latency is equivalent to 8192 register operations. The overhead for context-switching in non-blocking simulations is equivalent to 16 register operations. Whereas Figure 2 plateaued when the available parallelism was insufficient, this plot plateaus when the overhead (with respect to the latency) limits how much parallelism can be taken advantage of.

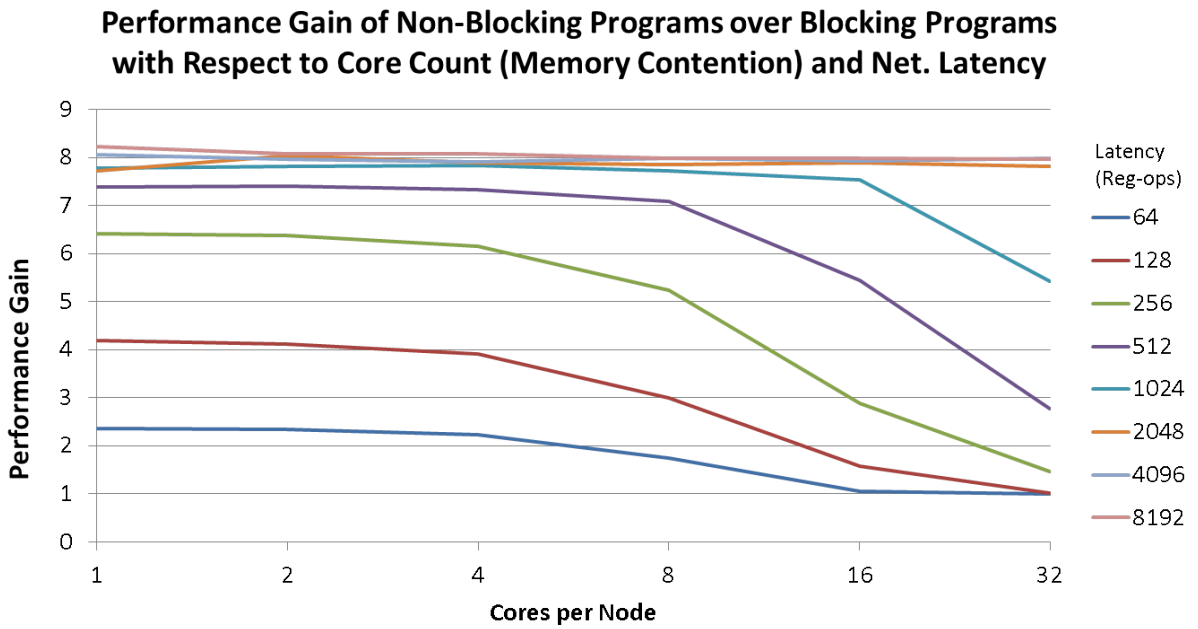

Figure 4. The performance gain of several simulations using non-blocking computations as compared to identical simulations that block computations upon accessing the network. All simulations were given an instruction mix consisting of 68% register operations, 8% network operations, and 24% memory operations. The overhead for context-switching in non-blocking simulations is equivalent to 16 register operations. Contention is also represented here; the likelihood for contention increases as the number of cores increases. For higher latencies,the probability of contention is lower, and its effects are less noticeable. Each core in the non-blocking simulations was assumed to have 8 tasks available for work. The number of tasks available for work (8) provides the upper bound for performance gain. The upper bound is slightly exceeded for higher latency cases on one core per node; the contributing factors to this are under investigation. This graph illustrates the trend that as network latency increases, the ability to overlap communication and computation also increases for non-blocking programs. The highest latencies maintain a relatively constant performance gain, consistent with Figure 2.

is that more cores present would cause a steady decrease in performance gain due to increased contention. Indeed, for most of the latency curves this behavior is observed. For the highest latencies, however, this effect is suppressed by the lack of available work (i.e. more tasks will spend more time on the network). This helps suppress the effects of contention because this decreases the probability that multiple tasks will be attempting to access the shared memory resource simultaneously. The sudden drop-off from the plateaus occurs at the point that the detrimental effects of contention become greater than the suppression of the performance gain due to starvation.

A portion of the results of the final set of simulations can be seen in Figure 5. The plot provided demonstrates trends with a constant latency of 1024 reg-ops. The results of simulations with different latencies all took on a similar appearance; higher latencies would merely shift the plot to the right while scaling up the performance gain.

This plot of this specific latency reveals several trends in the performance gain for non-blocking computational models. The first trend is the plateau in performance gain upon reaching a certain level of parallelism. This is the same effect as seen in Figure 3. Specifically, this plateau

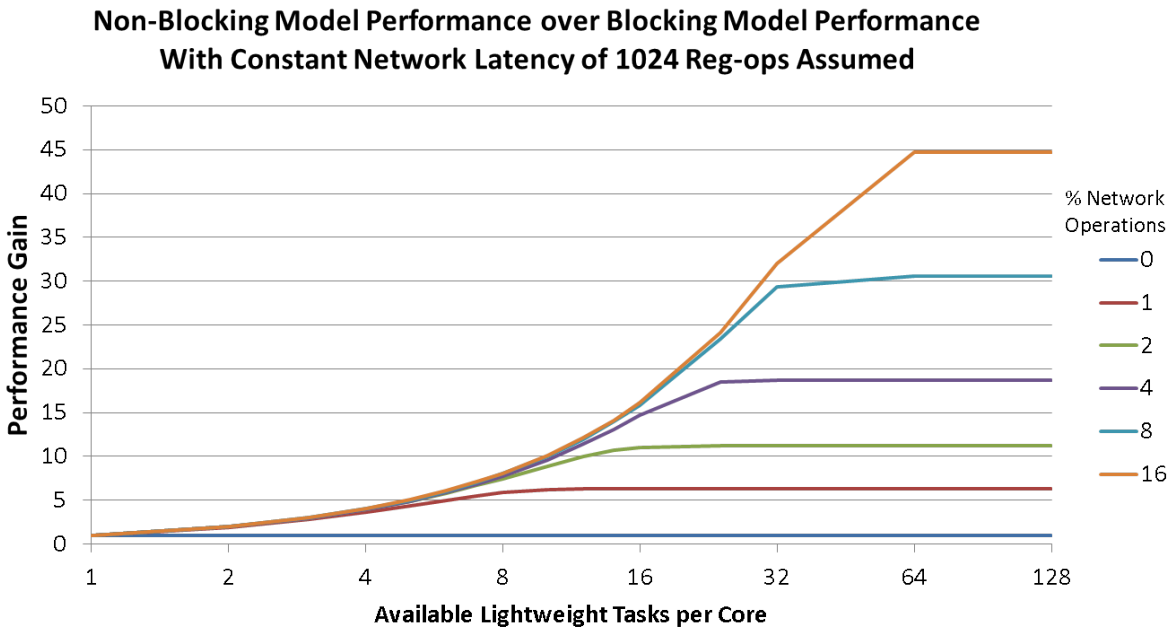

Figure 5. This figure represents the performance gain of a non-blocking program over a blocking program. Each graph plots the effects of changing the number of available tasks per core for various instruction mixes (each mix contains 68% register operations; the percentage of network operations is listed and the remaining percentage is memory operations). The network latency is equivalent to 1024 register operations representing a very fast network. The context switching overhead for non-blocking simulations is consistently 16 register operations. Only single core data were used for this plot. Each curve approaches an asymptote in performance gain near the point where the product of the number of available tasks and the context-switching overhead equals the network latency. This asymptote is consistent with Figure 3.

occurs when the product of the context-switch overhead and the number of available tasks is greater than the network latency. Even if all operations were network operations, at this point no further gain could be achieved by adding more available tasks to the system.

However, as indicated by the curves in Figure 5, there is an additional factor at play in determining where exactly this plateau begins. This factor is the instruction mix. For instruction mixes with fewer network operations, the gain curve appears to flatten earlier. This is because the probability that network operations will occur is low and thus the system is less likely to benefit from having more tasks ready to be run. In fact, it is the instruction mix that determines how rapidly the curves taper off to a plateau in all of Figures 2, 3, 4, and 5. Additionally, the instruction mix also determines the number of cores per node at which the contention for shared memory becomes noticeably detrimental. In general, however, it is the combination of the effects of starvation, latency, and overhead that determine the value of the plateau in performance gain.

6. How Do Major Execution Models Address SLOW

Table 1 examines several established execution models in terms of the SLOWER performance model but without energy and reliability (SLOW). Each entry of the table describes specific paradigm each execution model employs to addresses the corresponding source of performance degradation.

Table 1. A comparison of different execution models in terms of SLOW

	Starvation	**Latency**	**Overhead**	**Contention**
Sequential issue	Restricted to single instruction stream	Memory hidden by cache hierarchy	Avoidance; no parallelism to control	Only one request at a time
Vector	Fine grain scalar operations making up vector	Hide latency through pipelining	Amortize control across vector elements	Pipelined data flow movement
SIMD†	Data parallelism	Accesses of separate ALUs* to local data	Control amortized through broadcast	Overlap out of phase memory bank access
CSP‡	Process level parallelism	Coarse computation phases offset the effects of communication	Minimized by static scheduling	One request at a time for local data

† Single Instruction, Multiple Data.

‡ Communicating Sequential Processes.

* Arithmetic Logic Units.

As enabling technologies evolve, different approaches to their use must be developed to exploit the new opportunities they afford and to address the challenges they impose. The history of high performance computing is punctuated with a series of such paradigm shifts reflected by a succession of execution models, each responsive to the changes in technology properties as characterized by the factors of the SLOWER performance model. In the current decade of the post-VLSI (Very Large Scale Integration) era the technology demands system architectures comprising multiple cores per socket as the principal means of enhancing scalability either as a homogeneous array of cores or in compound throughput structures for specific favorable data flows. In addition to the need for medium grain parallelism, eventually at the billion-way level for the Exascale performance regime, the new architectures will require deep memory hierarchies, high and varying communication latencies, limitations on energy consumption, asynchrony of operation, and dynamic adaptive methods of resource management and task scheduling to exploit runtime information for improved efficiency. Other aspects of future execution models need to work effectively with lower average memory capacity per core and efficient lightweight communication access patterns.

A class of emerging experimental execution models has been evolving to address these technology challenges through the synthesis of a number of complementary semantic constructs. The ParalleX execution model [7], briefly discussed below, is representative of these models although it is, by no means, the only example.

ParalleX is an experimental many-tasking execution model that supports message-driven computation and uses dynamic adaptive methods to manage asynchrony and enable scalable operation on large systems. It achieves that by exploring synergistic effects arising from interactions of its primary semantic components, which include:

- *Locality* - an encapsulation of resources in a synchronous domain that guarantees bounded access and service request time as well as compound atomic sequences of operations;
- *Global Name Space* - provides the semantic means of accessing any first class objects in a physically distributed application;
- *ParalleX Processes* - an abstraction of distributed context hierarchy integrating data objects, actions, task data, and mapping data;
- *Compute Complex* - a generalization of thread concept;
- *Local Control Objects* - provide synchronization, management of parallelism, and migration of continuations;
- *Parcels* - implement message-driven computing that combines data transfer and event-based information using a variant of active messages to permit the management of distributed flow control in a context of asynchrony.

7. How ParalleX Addresses SLOW

The development of the ParalleX execution model is driven to a significant degree by the factors incorporated in the SLOWER performance model. ParalleX is intended, as were previous models before it, to mitigate the causes of performance degradation to improve both efficiency and scalability. In addition, ParalleX is being extended to respond to energy and reliability concerns. These latter two factors are not discussed in this paper to any extent and are only included for sake of completeness. Here each is considered in terms of the ParalleX strategy employed to mitigate it.

7.1. Starvation

Starvation is perhaps the principal limiting condition for extreme-scale computing. It is the insufficiency of concurrent available application work to fully utilize the physical processing resources. It is made more complicated by the distribution of the pending work across the separate system resources. Even with sufficient total concurrency, starvation can occur due to poor local allocation of work to resources. ParalleX addresses the challenges contributing to starvation by increasing the amount of parallelism available and facilitating work distribution. To expose and exploit more parallelism than conventional practices in a single unified model ParalleX incorporates coarse, medium, and fine grain parallelism semantics.

Coarse-grained parallelism is represented as ParalleX processes, which can span multiple localities (nodes), share nodes with other processes, and migrate if necessary. Processes provide the hierarchical framework for work and state dynamic distribution across system resources. Processes can have descendant processes in a dynamic tree of parent-child relationships, by which they provide coarse parallelism.

ParalleX organizes the actual work it performs as medium grain compute complexes, which often are performed as threads. The use of medium grained threads can expose much more parallelism than pure coarse grain models and permits context switching for non-blocking of physical resources. Local Control Objects provide declarative constraint based synchronization and scheduling to increase parallelism by permitting sophisticated control relationships and support dynamic adaptivity.

Fine grain parallelism is given in terms of static data flow control semantics within the context of a compute complex. This is a generalization of the myriad ad hoc approaches currently

used for Instruction Level Parallelism and permits translation to efficient forms of ILP for individual core architectures. Near fine grain parallelism is also exposed for usually remote compound atomic sequences of operations to exploit Remote Direct Memory Access (RDMA) techniques and smart networks.

Parallelism is sometimes simplistically described as either data parallelism or control parallelism. ParalleX treats these as aspects of a more general continuum in which a combination of these two can lead to runtime parallelism discovery based on meta-data for such structures as time varying irregular graphs. As discussed shortly, the reduction of overheads permitted by ParalleX allows finer granularity to be effectively exploited increasing the total amount of parallelism available, even for fixed size (strong scaled) applications. By providing powerful but focused synchronization methods (LCO), ParalleX largely eliminates over constraining global barriers, enabling overlap of successive phases of an evolving computation and exposing this additional form of parallelism. Together these principles embodied within the ParalleX model provide generality and flexibility both in size and form to maximize scalability.

7.2. Latency

Latency is the time for remote access and service requests resulting from distance of data movement. The primary effect of latency to performance is the blocking of the use of physical resources waiting for responses to remote data access requests and services. ParalleX addresses latency throught both introduction of mechanisms to reduce its magnitude and hiding its effects. ParalleX compute complexes are dynamic and can be exchanged in their use of physical resources. This context switching permits a thread that is waiting for a long latency request to be replaced by a pending task ready to do work, overlapping computation with communication. Multithreading of work trades parallelism for latency hiding. Parcels move work to the data, not always requiring data to be gathered to a fixed location of control. This allows migration of continuations (control state) to places where data access will be most intense, significantly reducing the latencies of the data accesses through the reduction of number and size of global messages. The use of local control objects provides mechanisms for event-driven control simultaneously addressing latency of action at a distance and managing the uncertainty of asynchrony of long latency operation.

7.3. Overhead

Overhead is the extra work performed to manage resources and support task scheduling. It is wasteful of time and energy compared to sequential execution. ParalleX within the constraints of hardware capability reduces the overheads and their effects found with conventional practices. It does this through the definition of focused control conditions, LCO, that support event driven computation control to minimize wasted work and avoid overly constraining semantics. Chief among these is the global barrier, which is almost all but eliminated through ParalleX. LCOs, benefitting from prior art, express rich semantics which has a powerful effect on parallel control state with minimum amount of overhead work. ParalleX encourages lightweight user threads in lieu of heavy weight OS threads (e.g., Pthreads) that are optimized to the needs of a specific application runtime to reduce the overhead of context switching time. To further avoid overheads, for very lightweight remote operations, ParalleX allows these to be specified and performed without incurring the overheads of compute complex instantiation.

7.4. Waiting Due to Contention

When more than one action requires access to a logical or physical resource at the same time, contention for the resource causes delays to all of the activities not given immediate access. ParalleX supports dynamic adaptive means by which, when viable, the runtime can allocate a different available resource with similar capabilities. Examples include allocation of memory banks, rerouting of network traffic where multiple paths exist, distribution of synchronization elements, multiple physical threads for task execution. The event-driven distributed flow control eliminates polling and reduces the number of sources of synchronization delays.

8. Conclusions

A performance framework has been introduced to facilitate development and optimization of extreme-scale abstract execution models and the future systems derived from them. SLOWER defines a six-dimensional design trade-off space based on sources of performance degradation that are derived from the physics of the underlying hardware systems and the control software. They are invariant across system classes in that they apply broadly and are shown here to invoke different system responses to changing enabling technologies. Looking forward, the new technology trends, which are leading to nano-scale and the end of Moore's Law, demand future innovations to address these same performance factors. An experimental execution model, ParalleX, is described to postulate one possible advanced abstraction upon which to base next generation hardware and software systems. A detailed examination is presented of how this class of dynamic adaptive execution models addresses each of the performance factors of SLOWER, opening a new path to highly efficient and scalable system design. There is not yet a unified formal analytical relation of the SLOWER performance framework, although it is an objective of this work to lead to such a theory. An alternative approach is to represent the trade-off space through a queue model and employ simulation methods spanning ranges of key parameters to expose the properties of the SLOWER framework. While this research path is still in progress, early results shown here demonstrate aspects of the parameterized design regime consistent with real world experience. These promising results also suggest a means to empirically derive the sensitivities of the interrelated factors. The immediate future work is to more extensively explore the SLOWER trade-offs through enhanced versions of the queuing simulation and use this to derive and validate an analytical formalism.

9. Acknowledgements

This work was supported in part by the Department of Energy under Award Numbers DE-SC0008809 and DE-NA0002377, and the Department of Defense High Performance Computing Modernization Program under Contract Number GS04T09DBC0017.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

2. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

3. G. E. Blelloch. *Vector Models for Data-parallel Computing.* MIT Press, Cambridge, MA, USA, 1990.

4. K. W. Cameron, R. Ge, and X.-H. Sun. logNP and log3P: Accurate analytical models of point-to-point communication in distributed systems. *IEEE Trans. Comput.*, 56(3):314–327, Mar. 2007.

5. C.-K. Chui. The LogP and MLogP models for parallel image processing with multi-core microprocessor. In *Proceedings of the 2010 Symposium on Information and Communication Technology*, SoICT '10, pages 23–27, New York, NY, USA, 2010. ACM.

6. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

7. C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling. Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model. *International Journal of High Performance Computing Applications*, 26(3):319–332, 2012.

8. R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, SC-9(5):256–268, 1974.

9. D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinational computation. *Annual Review of Computer Science*, 3(1):233–283, 1988.

10. F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A parallel computational model for synchronization analysis. *SIGPLAN Not.*, 36(7):133–142, June 2001.

11. R. Pasupathy, S. Kim, A. Tolk, R. Hill, and M. Kuhl. Open-source simulation software "JAAMSIM".