

‘ePX’ Cluster Supercomputing

James Glenn-Anderson, Ph.D. CTO *enParallel, Inc.*

Introduction

The essential basis of all *ePX*¹ technology is implementation of a true supercomputer processing model based upon multi-CPU/Graphics Processor Unit Array (GPA) architectural templates {18}{19}. In this configuration, the ***ePX* Desktop Super-Computer (ePX/DSC)** functions as a stand-alone PC-based scientific workstation for which processing performance is dominated by GPA order and the number of active thread-processors-per-GPU, (re: discussion of Amdahl’s Law in “***ePX Supercomputing Technology***” whitepaper {19}). Key advantages associated with *ePX* include; (1) parallel Single Instruction Multiple Data (SIMD) thread-processing employed by GPU’s facilitates broad-spectrum acceleration of algorithmic kernels commonly used in scientific computation, and (2) CPU/GPA *process-pipelining* (overlap) facilitates acceleration of complete applications well beyond capability associated with the standard CPU/GPU coprocessor model {19}. Theoretical analysis informed by comprehensive empirical testing suggests performance of a single *ePX/DSC* workstation will compare very favorably with small workstation clusters¹⁹ based on standard PC/CPU configurations. In simplest terms, this result is rooted in three considerations; (1) $O(10^1) - O(10^3)$ processing speed-up based upon an ‘ $N_{GPU} \times N_{TP/GPU}$ ’ acceleration factor, ($N_{GPU} \equiv$ Number GPU’s per GPA, $N_{TP/GPU} \equiv$ Number Thread-Processors per GPU), (2) essentially collision-free CPU/GPA interprocess communications based upon use of high-speed local interconnect, (e.g. ‘*PCIe*’, ‘*HyperTransport*’), and (3) CPU/GPA concurrency based upon non-blocking/asynchronous (‘streaming’) API transaction model.

It is interesting to note *ePX* advantages may be scaled to higher performance levels in essentially two ways; (1) *custom hardware* solutions based upon ever higher GPA order, and (2) *cluster-processing*, (i.e. adoption of a cluster-computing architecture). Each approach offers advantage of higher theoretical processing bandwidth. However, where physical space is not a significant issue, node clustering based upon standard PC form-factor remains a preferred approach to *ePX* performance scaling. This claim is motivated by; (1) relative ease with which a cluster may be implemented, (i.e. based upon availability of standardized API resources and COTS hardware components), and (2) realization of superior cost/performance ratio and scaling properties where the cluster approach is combined with the *ePX* supercomputer processing model. In particular, the custom hardware approach can engender significant difficulty and complexity; (1) significant incremental NRE cost, (2) challenging heat-transfer/cooling problems, (3) specialized API development effort, (4) local-interconnect total bandwidth constraint, and (5) local-interconnect form-factor constraint, (e.g. number of *PCIe* or *HyperTransport* slots). In what follows, we review key technical aspects of *ePX Cluster*¹ technology.

Cluster Architecture

The *ePX Cluster* architectural template is based upon more or less standard computer network communications infrastructure with *ePX/DSC* workstations¹⁸ placed at each processing node, (i.e. see *ePX/DSC* architectural template description in references {18}{19}). Two network communications standards are currently supported; (1) *Ethernet* {28}, and (2) *InfiniBand* {27}. ‘InfiniBand’ is based upon network switch **PHY**sical layer (PHY) with channel bonding features, while ‘Ethernet’ employs a far simpler CSMA/CD PHY. Of the two, InfiniBand exhibits far superior network scaling properties, (i.e. as one adds processing nodes). However, Ethernet is also far less expensive and available in form of commodity hardware components. Further, Ethernet has proven sufficient for a wide range of cluster processing applications. In what follows, Ethernet is assumed as the nominal *ePX Cluster* communications infrastructure, with understanding ‘Ethernet’ may be replaced with ‘InfiniBand’ for large-scale *ePX Cluster* solutions.

The fundamental concept for *ePX* supercomputer processing model implementation on clusters is parallel execution of component tasks distributed to multiple cluster-nodes, (i.e. see *processing model* discussion below). In *figure-1*, the nominal *ePX Cluster* architectural template is displayed schematically in block diagram form; component-tasks are distributed over an Ethernet-based network part and parcel of *ePX scatter-gather*. Cluster ‘*NODE₀*’ is further expanded in form of the *ePX CPU/GPA* architectural template. In this case, the network is hierarchically ‘flat’. However, the principle remains fully extensible to multi-cluster ‘grid’ networks, (i.e. exhibiting some form of network hierarchy). The *ePX* supercomputer model requires application level access to network communications infrastructure, (re: *network socket*-based interprocess communications), and a variety of standard TCP/IP-based {31} message-passing Applications Programming Interface (API) libraries may be employed for this purpose, (e.g. ‘*MPI*’ {15}{16}, ‘*OpenMPI*’ (Linux) {12}, and ‘*MPICH*’ (Windows) {17}).

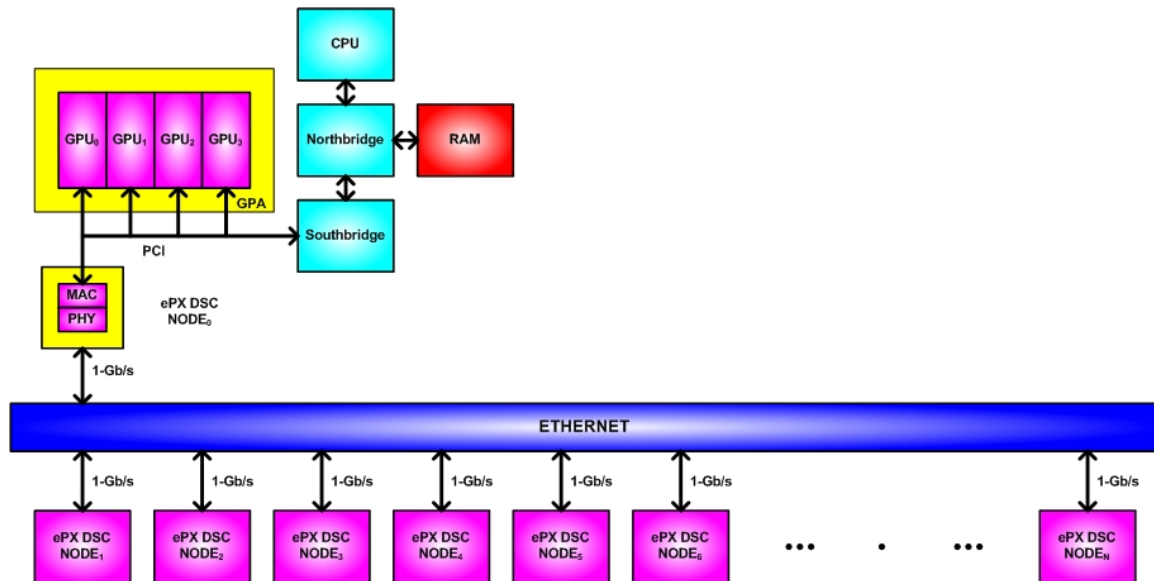


Figure-1: *ePX Cluster* Architectural Template

Processing Model

Workstations based upon CPU/GPU-coprocessor architectural templates have already been clusterized for extended performance scaling {2}{9}. However, *ePX Cluster* extends this idea so as to realize all advantages derived from implementation of the *ePX* supercomputer processing model at all cluster-nodes, (i.e. *ePX/DSC* at each processing node). In particular, aside from the raw acceleration potential inherent a GPU array, it will be shown *ePX Cluster* employs a highly flexible *task-kernel-thread* process hierarchy by which; (1) spatio-temporal data coherence properties may be exploited cluster-wide, and (2) I/O constraint boundaries are more easily placed at associated processing performance constraints for realization of highest performance potential.

ePX Cluster scaling is achieved by virtue of an enhanced parallelism applied to *dataflow*² structures larger than what might be parallelizable on a single *ePX* processing node. In effect, the *ePX* supercomputer processing model is extended to a *task-kernel-thread* application decomposition, with *task* processing distributed to cluster-nodes part and parcel of GPA *scatter-gather* operations already applied to *kernel* and *thread* components. A critical subtlety emerges in that total *ePX* node processing bandwidth admits mapping of generally larger task-components when compared with standard PC nodes at a constant inter-node communications envelope, or an equivalent total processing time. Assuming perfect load-balancing and I/O constraint boundaries placed at the associated multi-CPU/GPA processing performance constraint boundaries, (i.e. no processing element stalls for lack of datapath; see *Amdahl's Law* discussion below), highest performance potential is realized at a total cluster acceleration (A_{TOTAL} ; ' N_{NODE} ' \equiv Number cluster processing nodes) given by:

$$A_{TOTAL} = N_{NODE} \cdot A_{NODE} \quad (1)$$

At perfect multi-CPU/GPA pipelining, (re: *Amdahl's law* discussion in {19}), over all cluster nodes, theoretical maximum *ePX Cluster* performance potential is given by, (N_{GPU} \equiv Number of GPU instances in array, ' $N_{TP/GPU}$ ' \equiv Number of thread processors per GPU):

$$A_{TOTAL} = N_{NODE} \cdot A_{NODE} \cong N_{NODE} \cdot N_{GPU} \cdot N_{TP/GPU} = N_{NODE} \cdot N_{GPA} \quad (2)$$

Thus, we observe a triple-axis scaling relation unique to *ePX Cluster* - linear scaling in the number of cluster processing nodes, number of GPU's employed at a (cluster) processing node, and number of parallel thread processors per GPU. One might surmise scaling may be performed more or less equivalently along any axis and to some extent this is true. However, this analysis engenders implicit assumption of collision-free inter-node communications characteristic of parallel and tightly coupled cluster processing. Thus, at any collision-free constraint boundary, cluster node I/O bandwidth must proportionally decrease as one increases ' N_{Node} '. This requirement may be satisfied with processing of sufficiently large component tasks at each node, at constant-acceleration³. Assumption of a constant-acceleration constraint in turn implies concomitant scaling in ' N_{GPA} '. Thus, the aforementioned triple-axis scaling relation is understood as

fundamentally vectorial. In particular, cluster performance scaling hinges upon the fact *ePX* optimized CPU/GPA performance facilitates reduced internode communication bandwidth.

An example application dataflow¹⁵ with superimposed component task partition is displayed in *figure-2*. Noting communications internal to any given component task are local to a single cluster-node, (i.e. don't appear in the internode communications budget), it can be shown internode bandwidth requirements may be controlled at associated internode I/O boundaries with adoption of simple conditions on equivalent subgraph structure^{3,4}. Qualitatively, the number of algorithmic kernels comprising a given component-task, (i.e. 'size'), determines internode bandwidth overhead and the number of CPU/GPA resources available to a given component-task determines performance at a given node. This essential relationship is then leveraged as basis for optimizing cluster performance at the internode communications constraint boundary, (re: *equation-2*).

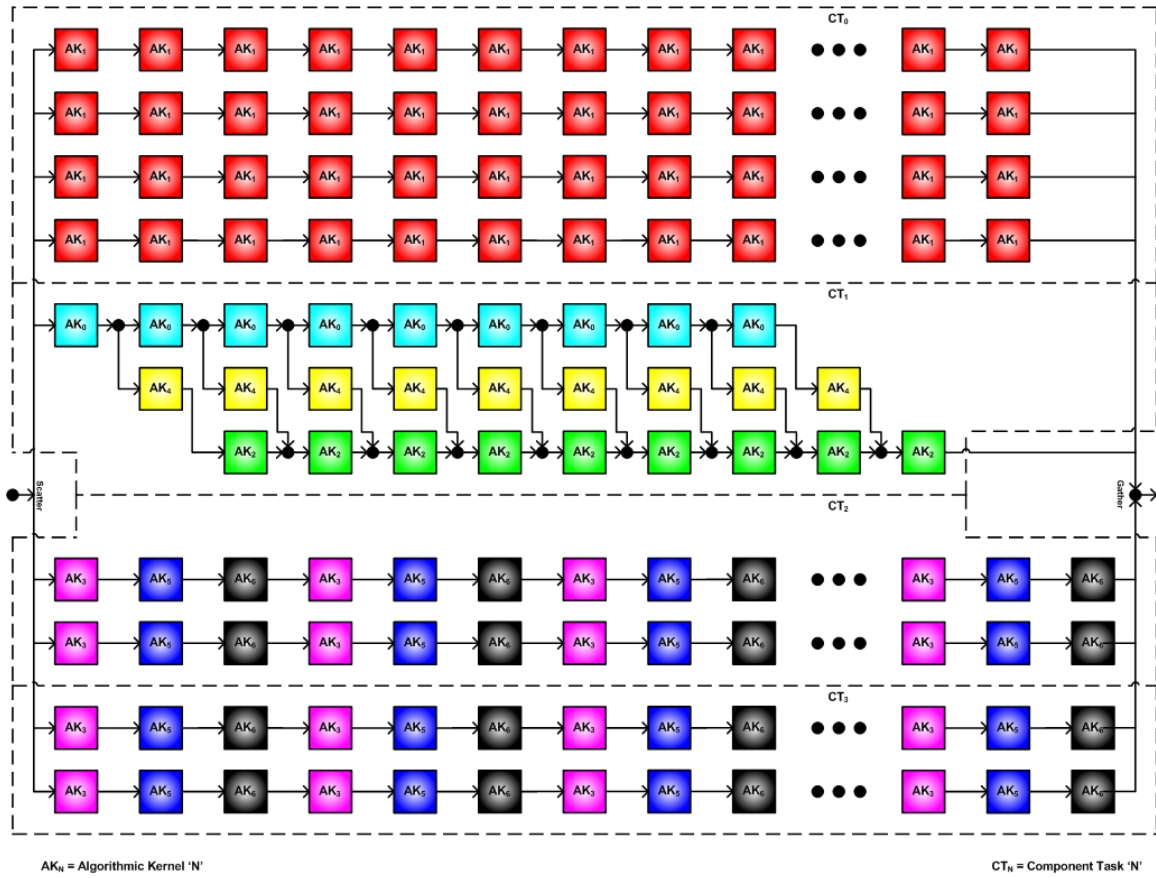


Figure-2: Example DataFlow Component-Task Elaboration

ePX Cluster extends the supercomputer processing model described in “*GPU-based Desktop Supercomputing*” {18} and “*ePX Supercomputing Technology*” {19} to *scatter-gather* distribution of processing threads across the cluster infrastructure. In effect, a hierarchical schema is employed whereby application component-tasks are distributed to

processing nodes, algorithmic kernels are distributed to GPUs, and threads are processed at each GPU according to a parallel SIMD (vector) model. Thus, hierarchical software decomposition and parallel-processing at each level of hierarchy is implied throughout.

The *ePX* supercomputing model is distinguished by optimal scheduling against GPA processing resources. In particular, algorithmic kernels are dynamically mapped to GPU instances (scheduled) based upon; (1) GPU-element availability and (2) opportunistic SIMD instruction pipeline reuse. In this manner SIMD **C**yclostatic **T**hread **R**esidency (*'CTR'*; see discussion in ref. {19}) is effectively maximized at any GPU instance, affording highest possible acceleration efficiency cluster-wide. In present context, *'CTR'* is defined as a measure on the expected proportion of time during which the instruction pipeline is performing actual datapath calculations, (e.g. as opposed to device I/O, instruction pipeline initialization, and thread synchronization). Associated *scatter-gather* distribution of *work-units* consisting of processing threads and any associated datapath to GPA resources is performed according to scheduler state. A given thread-set may be applied to a GPU instance at initialization or may already exist in situ as result of a previous processing cycle. In the latter case, the scheduler will opportunistically forego pipeline re-WRITE/initialization, (re: *instruction pipeline reuse*), and apply only datapath during a given *scatter* cycle. In this manner, algorithmic kernels are parallelized at the GPA transaction buffer and thread-sets optimally processed in parallel within GPU/SIMD instruction pipelines. This bipartite parallelism critically depends upon the fact *scatter* at the GPA transaction buffer is *non-blocking*. Thus, the CPU does not have to wait for completion of a GPU processing cycle. In this manner, CPU/GPA thread processing may be effectively overlapped. Note *gather* remains blocking according to the associated dataflow representation and implied scheduler synchronization semantics. *ePX Framework* further implements all required *scheduler*, *scatter-gather*, and *CPU/GPA pipelining* management functionality based upon an abstraction by which work-unit structure and interprocess communications implementation details are effectively hidden. In effect, all such details are pushed to process-queue service routines. Thus, *ePX management* operations remain generic across all *multicore-CPU/GPA* and derivative *cluster* architectural templates regardless of the specific nature and location of process components.

At any *ePX* node, distinct *scatter-gather* process queues are maintained for each mapped processing resource. Service routines attached to these queues are responsible for *work-unit* WRITE/READ transactions at associated buffers. *ePX Cluster* employs three such buffer classes and associated methods corresponding to GPA, CPU, and NODE (cluster) resources. Multi-CPU/GPA transaction sequences are already described in “*ePX Supercomputing Technology*” {19}. NODE transactions are mitigated by an associated interprocess-communications API. Depending upon application requirements, Operating System (OS), and architectural template, a variety of (interprocess communications) API's are supported; (1) MPI {13}{15}{16}{17}, (2) OpenMPI {12}, and (3) OpenMP {14}, (i.e. see '*ePX Middleware*' discussion below). The aforementioned *scatter-gather* service routines then implement API-specific calls for transparent access to a given processing resource. In standard configuration, *ePX Cluster* employs three distinct APIs

for this purpose; (1) *CUDA* (GPA) {4}, (2) *OpenMP* (multi-core CPU), and (3) *MPI* (cluster-node).

Nominal *ePX Cluster scatter-gather* pathways are displayed in *figure-3*. Component tasks are placed at cluster nodes as *daemons*, or spawned via '*FORK-EXE*' and propagated over the cluster network, (i.e. see *ePX Middleware* discussion below; '*MOSIX*' references {23}{24}{25}). In this specific case, component-tasks originate at *ePX/DSC-NODE_N* and are propagated to all other nodes. At *NODE₀*, distribution of algorithmic kernels to GPA elements, (i.e. scatter-gather), and associated subtasks to multicore CPU, (i.e. multithreaded), are also displayed.

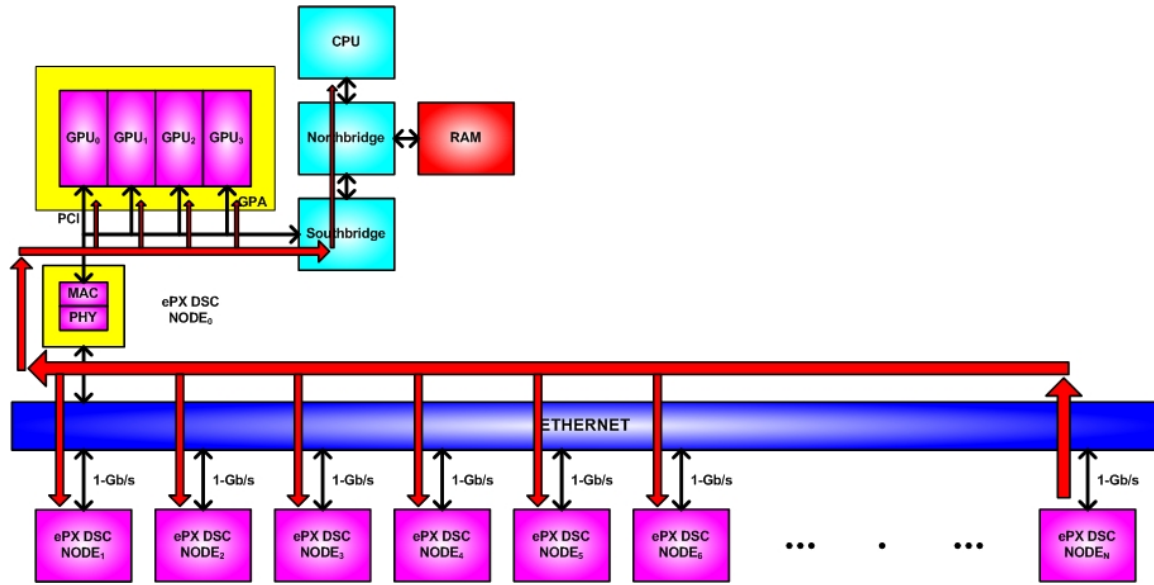


Figure-3: *ePX Cluster Scatter-Gather Pathways*

Within context of any application, all *ePX Cluster* nodes remain fully capable of performing *scatter-gather* on any unused cluster resource. Thus, *dataflow-object parsing*, *schedule generation*, *work-unit composition* and *scatter-gather* are all distributed according to the *task-kernel-thread* hierarchy. In this manner, all cluster resources may be brought to bear in highly flexible manner, with maximal parallelization¹⁰ and minimal impact upon internode communications resources¹¹. Application execution is initiated at a single control node with discovery and registration of all available cluster nodes followed by schedule generation. Application processing proceeds with *work-unit* distribution to peripheral cluster resources per the schedule part and parcel of the assumed supercomputer processing model; associated datapath may be propagated as an explicit work-unit component or virtualized based upon data-server transactions at the originating cluster-node, (i.e. part and parcel of dataflow elaboration). Upon receipt of a *work-unit* at a peripheral cluster-node, the resident *ePX-manager* initiates discovery and registration of available cluster resources, followed by generation of a local schedule, followed by *scatter-gather* more or less identical to that performed on the originating node. A key subtlety is the descriptive term '*component-task*' is understood as referring

to processing of a specific *dataflow-object* and *application components (software)* present at any associated *cluster-node* exist as copies of the code running on the *originating cluster-node*. This has significant ramification for both *ePX* software design and *FORK-EXE*-based (component) task distribution {23}{24}{25}.

Scheduler

ePX-scheduler establishes a basic organizational schema for all *scatter-gather* operations on cluster processing resources according to process dataflow. Where *ePX Cluster* is considered, an initiating node performs dataflow elaboration, followed by schedule generation, followed by *work-unit* assembly and distribution (*scatter-gather*) onto the cluster. Cluster schedule optimization is based upon minimization of total processing time subject to maximum memory bandwidth and interprocess bandwidth constraints. An immediate consequence of this constrained optimization is *minimal-processing-time* is generally not equivalent to *maximal-parallelization*. At the performance boundary, a trade-off is established between speed-up due to parallelization and degradation due to associated communications overhead, (re: contention-free access to processing resources). This trade-off is managed by synchronized propagation of control, instructions, and datapath over communications infrastructure based upon a tightly-coupled process schedule. Where *ePX Cluster* is considered this essential trade-off is rooted in the effectively ‘flat’ nature of cluster-node communications infrastructure.

An example *ePX Cluster-node* component-task dataflow is displayed in *figure-4*. This dataflow is understood as being generated part and parcel of dynamic process-scheduling. In present context, ‘dynamic scheduling’ is seen to imply presence of (process) conditionals at an originating cluster-node by which dataflow objects are generated, parsed, and distributed to cluster-node resources. Conditionals associated with task/kernel synchronization, CPU process pipelining, and instruction pipeline reuse, (re: are updated at each processing cycle according to dataflow elaboration. Once generated, dataflow processing is effectively identical at all cluster-nodes. In *figure-4*, component-tasks are color-coded based upon mapping to local resources, (i.e. ‘yellow’) or distributed via *scatter-gather* to another cluster-node, (i.e. ‘blue’).

While any of a multitude of approaches of varying complexity may be employed for scheduler design, *ePX-scheduler* is currently based upon a particularly simple ‘greedy’ heuristic affording robust and near-optimal performance; “*Everything that can be scheduled is scheduled*”. The scheduler algorithm is also considered *distributed-recursive* in the sense that; (1) any *dataflow-object* scattered onto the cluster will necessarily lead to scheduler invocation under aegis of an *ePX-manager* copy, albeit on a distinct processing node (*distributed*), and (2) the original dataflow is hierarchically disassembled and processed piecemeal, (i.e. ‘divide and conquer’), by a flow-dependent sequence of scheduler invocations (*recursive*); in this instance full recursion semantics⁷ accrue based upon the ‘call-back’ property implicit to *scatter-gather*.

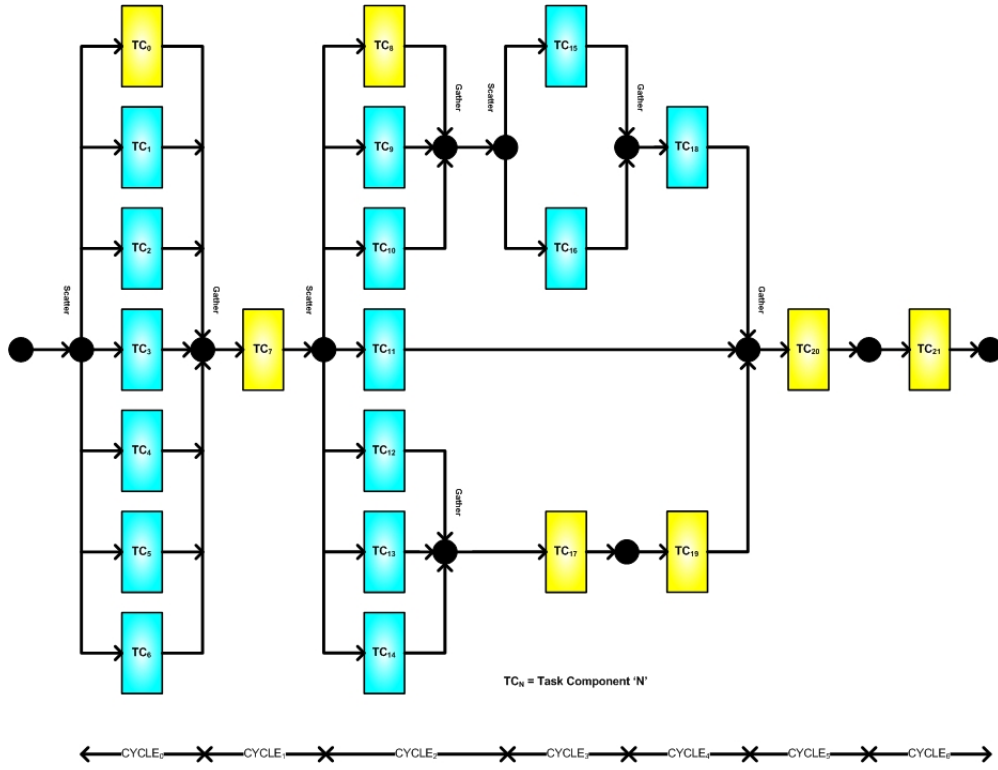


Figure-4: *ePX Cluster-node Component-Task DataFlow Example*

Cluster internode communications are typically based upon InfiniBand⁵ {27} or Ethernet⁶ {28} PHY; InfiniBand represents a favored cluster/supercomputer interconnect technology due to availability of a scalable-bandwidth PHY, while Ethernet remains far more ubiquitous and relatively low-cost. However, in present context, relative advantages and disadvantages are obviated in face of the fact each engenders some contention-free bandwidth constraint capable of significantly impacting cluster performance. An essential point is collision/contention resolution remains a fundamentally statistical process with ancillary overhead; internode communication must then be considered ‘lossy’ with result expected (internode) communication rates remain less than maximal. Thus, statistical limits are placed upon the degree to which component task processes may be tightly coupled. This effect is displayed in *figure-5* where component-task time intervals are padded with expected statistical wait (‘**O**verhead’) intervals. As shown by the expanded CPU/GPA task schedule for CT_0 , all kernel and thread execution intervals associated with that task are then similarly padded. In this manner, performance at all levels of processing hierarchy is effectively reduced. Further, at fixed equivalent internode bandwidth, these effects tend to increase as cluster nodes are added, (re: cluster scaling properties), with result intertask coupling and overall processing efficiency is further degraded. In this manner, the aforementioned *task-parallelization/communications-overhead* performance optimization trade-off emerges as a cluster scaling property.

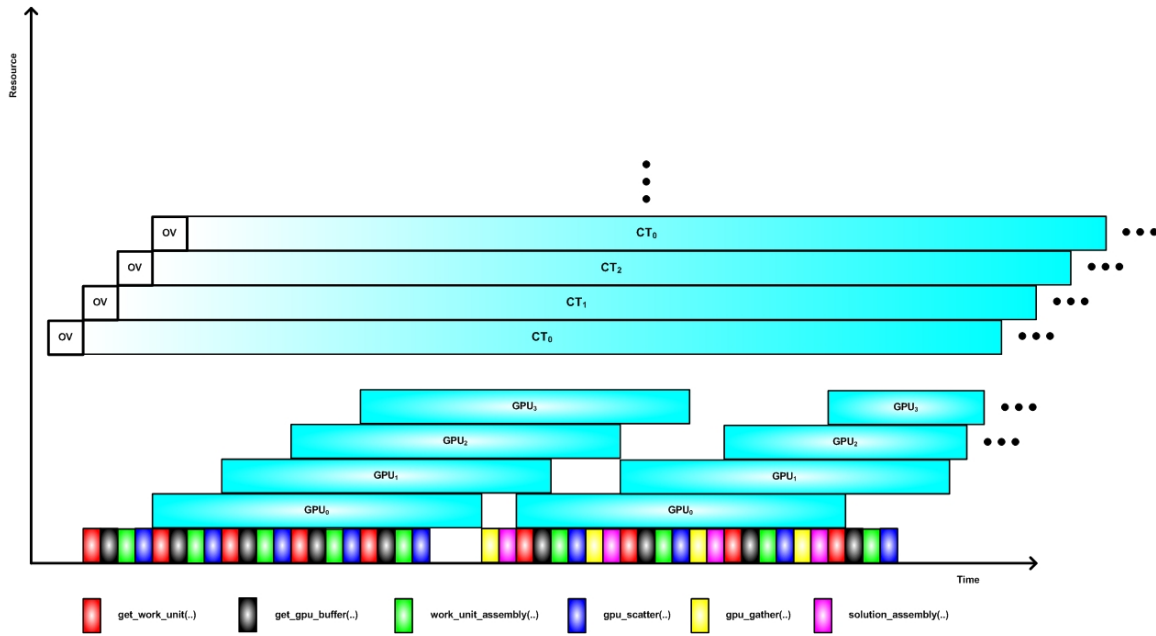


Figure-5: *ePX Cluster* Task-Schedule Example

Scatter-Gather

The *distributed-recursive nature* of *ePX-scheduler* then implies associated *scatter-gather* operations will also exhibit a distributed-recursive pattern. In effect, any *ePX-manager* instance parses an input dataflow-object, schedules processing of task-components on some combination of (local) multi-CPU/GPA and cluster-node resources, and then scatters all process components. Kernels are processed locally to the limit of available GPA resources and the remainder composed as task-components and scattered to available cluster-node resources. This process is then repeated at any receiving cluster-node part and parcel of distributed-recursive scatter-gather. In this manner, all cluster resources may be brought to bear based upon the assumed *task-kernel-thread* hierarchy and *ePX* supercomputer processing model. A critical advantage is realized whereby mapping of software components to processing resources is rendered highly flexible. This flexibility is expressed in form of specific parametric dependencies commonly associated with a full-featured supercomputer processing model, (e.g. parallelization hierarchy/order, work-unit composition/order, and process pipeline depth). A further advantage is gained in that internode communications are greatly reduced relative to an approach whereby all *scatter-gather* is performed at a central control-node, (i.e. initiating node).

The *ePX* supercomputer processing model critically depends upon non-blocking ‘scatter’ at all transaction buffers, as implemented by the *GPU*, *multi-CPU*, and *cluster-node* API’s. This feature enables overlap of GPU processing at all array elements and pipelining of cluster-node and multi-CPU/GPA processes. The result is an effective concurrent process schedule by which true supercomputer performance may be realized. On the other hand, ‘gather’ operations throughout the dataflow remain blocking, as

defined by (local) scheduler synchronization semantics. Scheduler synchronization is defined according to dataflow topological, (i.e. '*graph theoretic*'), structure by which specified inputs must be concurrently present in order for processing to proceed along some dataflow branch. Where *ePX Cluster* is considered, synchronization events are resolved at distributed *ePX-manager* instances.

'*Scatter-Gather*' work-unit assembly is based upon a dynamic dataflow design representation combined with an algorithmic kernel decomposition created part and parcel of *ePX* applications software development methodology. In particular, *ePX* software design is fundamentally expressed as a global dataflow-object with tasks mapped to cluster nodes, algorithmic kernels mapped to GPU instances, and threads mapped to GPU thread processors. Concurrent kernels nominally express *data-parallelism*⁸ constrained by available GPU device memory and processing resources and optimally sized based upon; (1) SIMD cyclostatic residency, (2) GPA load-balancing, and (3) SIMD instruction pipeline reuse. Concurrent tasks nominally express both *task-parallelism*⁸ and *data-parallelism*; with expectation data-parallel elements will ultimately be extracted and mapped to local multi-CPU/GPA resources.

Multicore CPU/GPA Pipeline

Where the *multi-CPU/GPA* architectural template is being considered, control is returned to the CPU after any non-blocking *scatter* operation, (i.e. see *CPU/GPA Pipeline* discussion in {19}). While function calls at the CPU are nominally processed sequentially according to some variant on the Von Neumann model, use of multicore CPU admits multi-threaded processing of kernels mapped to the CPU process-queue. While scatter-gather is not employed, multithreading is parallelized based upon an assumed Symmetric Multi-Processing (SMP) model, with full Operating System (OS) and runtime-system support implied. In effect, the pipelined CPU code components in CPU/GPA {18}{19} may be regarded as replaced by a third axis of parallel-processing in *multi-CPU/GPA*. In this context, CPU code parallelization is by definition 'local' - total parallelization may be increased with little or no increased internode communication. Of course, at any given node-instance, some non-parallelizable code fraction necessarily remains¹². However, the generally shorter CPU-process timelines associated with multicore facilitates improved local CPU/GPA pipelining and generally improved global performance. In present context, multicore CPU parallel-processing model implementation is greatly facilitated by; (1) *ePX* use of generalized process queues, and (2) availability of standardized (multicore) API libraries, (e.g. '*OpenMP*' {14}, '*OpenCL*' {33}, and '*CUDA*' {29}), as basis for associated process-queue service routines. From a global cluster-processing perspective, Amdahl's Law still holds, albeit in a slightly different way, (i.e. see *Amdahl's Law* discussion below).

Kernel Dependencies and Composition

Optimal *kernel* processing part and parcel of the *ePX-Cluster* processing generally results under conditions where mapped kernels remain fully *encapsulated*, and *data-parallel*⁸. In this context, an *encapsulated kernel* is self-contained; all required data and instructions

are available at SIMD pipeline initialization. Thus, once the instruction pipeline is initialized, no ancillary interprocessor communication, (i.e. ‘*message-passing*’), or thread resynchronization is required for completion of a processing cycle. Additionally, *data-parallel* kernels are typified by a single I/O process and single SIMD pipeline initialization during any processing cycle. Conversely, a *task-parallel*⁸ kernel is actually a ‘composition’ of multiple *data-parallel* kernels, each of which requires a distinct I/O process and SIMD pipeline initialization during any processing cycle.

Generally speaking, any use of non-encapsulated kernels in an application dataflow necessarily implies additional CPU processing and resynchronization across CPU/GPU process instances; any increased CPU processing gives rise to increased CPU/GPA pipeline overhead and GPU resynchronization reduces equivalent SIMD processing efficiency. However, there exist important circumstances requiring use of non-encapsulated kernels; (1) dynamic *scatter-gather* control, and (2) passing intermediate results between GPA elements. In the former, global state is updated based upon kernel processing output and the process dataflow modified according to conditionals in turn based upon global state. Once activated, any dataflow-object thus generated is *scattered* as described above¹³, (re: ‘processing model’ discussion). In the latter case, a given kernel too large for execution on a single GPU is split into multiple *fractional-kernels*, effectively requiring use of a Symmetric Multi-Processor (SMP), (i.e. shared-memory), processing model {30}. This scenario may be further complicated by distribution of fractional kernels across multiple cluster nodes. Difficulties emerge in that the cluster architectural template is characteristic of a distributed memory system; while the message-passing interface already present at each cluster-node can be leveraged for creation of a virtual shared-memory image, considerable performance overhead may accrue based upon ‘*MUTEX*’ conditions required for guaranteed memory consistency. Nevertheless, while appearance of any overhead is undesirable it is also important to note use of non-encapsulated kernels provides additional resource mapping options. In fact, some applications will require this feature in order to satisfy hardware resource constraints or otherwise optimize the process schedule.

Load-Balancing across all mapped cluster-resources generally facilitates mutual synchronization of component processes as basis for overall performance optimization. As point of fact, process kernels generated part and parcel of dataflow elaboration will accrue in a variety of ‘sizes’. Thus, an important optimization technique emerges in form of optional kernel composition, (re: ‘*task-parallelism*’), at a GPU process-buffer; a number of small kernels may be composed in form of a single work-unit, (i.e. equivalent to a ‘*task-parallel*’ kernel), and applied to the GPA in a single *scatter* operation. However, each associated instruction pipeline initialization¹⁴ and I/O process still serve to reduce SIMD processing efficiency. In this manner, another essential performance optimization trade-off is revealed in form of GPU ‘*load-balance versus efficiency*’.

An essential point is, despite incurred overhead, both *non-encapsulation* and *task-parallelism* remain useful as design options. Further, any valid process schedule optimization must necessarily include overhead associated with kernel dependencies and *work-unit* composition.

ePX Middleware

The *middleware* software component provides access to processing resources via a standardized set of Application Programming Interfaces (API), based on a generic software framework development model. These API components correspond to; (1) cluster-node, (2) multicore CPU, and (3) GPA processing resources accessed by service routines attached to *ePX* process queues. Each API provides a hardware-abstracted transaction model in support of non-blocking *scatter-gather* plus ancillary *interprocess communication*, (i.e. including local GPU-GPU transactions). In this manner, *ePX Cluster* is understood to employ a hybrid parallel-processing model by which concurrency at cluster, multicore, and GPA levels of architectural hierarchy is achieved.

cluster-node API is used to distribute and manage component tasks across cluster infrastructure consisting of some set of processing nodes and an accompanying internode network communications resource. In particular, this API is leveraged to overlap network communication with CPU/GPA processing based upon a global communications schedule. Where *ePX Cluster* is considered, cluster-node API is nominally based upon the Message Passing Interface standard {15}{16} and may occur in one of two specific forms; (1) ‘*OpenMPI*’ (Linux) {12}, and (2) ‘*MPICH*’ (Windows) {17}. The basic MPI version 1.3 implements a distributed memory model with; (1) TCP/IP-based interprocess communications, (2) flexible synchronization semantics, (3) essentially virtual process topology, (e.g. with use of ‘point-to-point’ rendezvous and ‘graph’ process representation), (4) process-pair exchange, (5) synchronous/asynchronous [multiple operations]) operation, and (6) generic C/C++ bindings all within context of a static runtime environment. MPI version 2.1 extends these features in support of; (1) shared-memory model, (2) parallel I/O, (3) dynamic process management, and (4) remote memory operations.

multicore API is then used to access multithreaded parallel-processing capability of multi-CPU processor architectures, (e.g. ‘quad-core’ Intel/AMD processors) - *ePX Cluster* currently supports two specific API’s for this propose¹⁶, (i.e. depending upon specific hardware configuration); (1) ‘*OpenMP*’ (nominal) {14} and (2) ‘*CUDA*’ {29}. The multicore API implements a multithreading, shared-memory multiprocessing model, (re: **Symmetric Multi Processor (SMP)** model), for CPU code components residing at a given cluster node. In this case, the multithreading feature is enabled via use of ‘**FORK-EXE**’ for creation of slave-threads that may in turn be propagated to any available CPU core and subsequently executed ‘in-parallel’. The *ePX* supercomputer processing model leverages this feature to optimize multicore processing based upon a *locality-of-reference* property implicit to the *task-kernel-thread* hierarchy discussed above. In this context, a run-time environment is established by which CPU processing threads may be allocated to different processors (i.e. in form of ‘work-sharing’ constructs; compiler directives + environment variables are leveraged to influence run-time behaviors).

GPA API is used to access GPU array I/O, (re: generic *ePX* GPA process-buffer service routines), memory management, (re: GPU device/shared memory, parallel cache), and thread management functionality, (re: SIMD process model), in form generic

programming function calls, (e.g. C, C⁺⁺, Java, Python). In this manner, details associated with GPA and architectural template hardware implementations are effectively hidden within context of *ePX* scatter-gather, SIMD instruction pipeline initialization/reuse, and datapath I/O operations. Further, *ePX* confinement of all GPU/SIMD thread-programming detail to GPU-specific function calls enables three important advantages; (1) *ePX* code portability, (i.e. generic to level of ‘include’ libraries), (2) simplified, (i.e. essentially independent), optimization of cluster-node, multicore, and GPA code components, , and (3) *ePX acceleration library* code-encapsulation. GPU APIs currently supported by *ePX* include NVIDIA’s Compute Unified Device Architecture (‘*CUDA*’) {4}{5}{6}{7}{29} and ATI’s Data Parallel Virtual Machine (‘*DPVM*’) {11}. Standard compiler support is also provided for all GPU-resident code.

ePX Framework Pseudo-Code

The *ePX* software framework is extended to include functionality associated with *dynamic-dataflow* elaboration and *dataflow-object* parsing part and parcel of the previously discussed cluster processing model. The resulting structure is abstracted in form of the simple ‘C-esque’ *p*-code component displayed below. In this case, given the dynamic nature of data structures to which processing is applied, a fundamental *object-oriented* code representation is employed. In essence, all *elaboration*, *scheduler*, *scatter-gather*, and *pre/post-process* functionality is combined under a nested iteration loop polling completion of all function components. At start of processing, ‘*elaborate(..)*’ generates a new dataflow-object based upon processing of; (1) an existing design representation object, (i.e. ‘*originating-node*’), or (2) a *work-unit* already residing on the cluster process queue, (i.e. ‘*daughter-node*’). In either case, a *dataflow-object* pointer is created and passed to the inner (local) processing loop for *resource-scheduling* and *scatter-gather*. At this level, processing proceeds sequentially through all functional components. However, a key subtlety associated with *ePX* framework is revealed in the fact although function invocations are performed sequentially at the CPU, associated operations are performed with effectively arbitrary order, number, and concurrency, as determined by the process schedule. This enables use of a wide variety of control paradigms, based upon diverse optimization schemes, applied resource constraints, and desired schedule properties; parallel-process scheduling remains a matter of design choice, as long as synchronization and completeness requirements are satisfied.

```

//
//      Application 'Front-End' ...
//
.
design_structure          design;
dataflow_structure      *dataflow_pointer;
scheduler_structure     *schedule_pointer;
.
.
.
do
{
    elaborate(...dataflow_pointer,...,design,...,&flag_elaborate,...flag_origin,...);
    do
    {
        scheduler(...dataflow_pointer,...,schedule_pointer,...);
        preprocess(...schedule_pointer,...);
        scatter(...schedule_pointer,...);
        message(...schedule_pointer,...);
        gather(...schedule_pointer,...);
        postprocess(...&flag_process,...,schedule_pointer,...);
    }
    while (flag_process);
}
while (flag_elaborate);
.
.
//
//      Application 'Back-End' ...
//

```

<i>ePX</i> Component Function	Description
<i>elaborate</i> (..)	Creates and instances <i>dataflow-objects</i> for local and cluster scatter gather; (1) if ' <i>originating-node</i> ', new <i>dataflow-object</i> is created based upon elaboration of dynamic dataflow design representation, (2) if ' <i>daughter-node</i> ', new <i>dataflow-object</i> is created based upon received work-unit residing on cluster process-queue, (3) flags completed elaboration over design representation object.
<i>scheduler</i> (..)	Dynamically parses dataflow-object independently along all branches according to <i>component-task/algorithmic-kernel</i> hierarchy and <i>scatter-gather</i> semantics as expressed by associated graph-theoretic structure. Cluster/CPU/GPA processing calls are indexed on a <i>schedule</i> object. All supercomputer processing control is then dynamically generated based upon contents of reserved <i>schedule-control</i> fields. Selected <i>schedule-state</i> components are also placed in reserved data fields and leveraged within context of process control.
<i>preprocess</i> (..)	Assembles work-unit instances for scheduled processes in form of pointers to; (1) available cluster-node/GPU-instance, (2) function to be executed, and (3) associated datapath. Where SIMD pipeline reuse is performed, <i>schedule-state</i> is examined to determine if the associated function is already present. Completed work-unit assembly is then updated on reserved <i>schedule-state</i> fields, subsequently used to conditionally initiate <i>scatter</i> .
<i>scatter</i> (..)	Applies work-units to mapped resources at the appropriate transaction interface; (1) generates all associated WRITE operations at transaction buffers, (re: <i>node/kernel-scatter - non-blocking</i> return), and (2) updates <i>schedule-state</i> .
<i>message</i> (..)	Manages GPU/GPU - CPU/GPU – NODE/GPU message-passing or exchange of intermediate datapath based upon interprocess communication permissions as defined by associated <i>schedule-control</i> field; (1) formats message/datapath datagrams, (2) generates associated READ/WRITE operations at GPA transaction buffer, and (3) updates CPU-resident conditionals, (e.g. associated with datapath elaboration).
<i>gather</i> (..)	Applies (blocking) synchronization lock at dataflow 'gather', as defined by associated <i>schedule-control</i> field; (1) generates all associated READ operations at transaction buffers, (2) assembles intermediate results for subsequent post-processing, and (3) updates <i>schedule-state</i> .
<i>postprocess</i> (..)	Completes processing at dataflow 'gather' as defined by associated <i>schedule-control/state</i> fields; (1) applies post-processing operations to complete set of intermediate results, (2) stages result for subsequent work-unit assembly or cluster WRITE-BACK, (3) updates <i>schedule-state</i> , and (4) flags completion of all processing on dataflow, (re: ' <i>flag_process</i> ').

Table-1: *ePX* Framework Component Functions ('p-code')

Amdahl's Law

As indicated in previous discussion of *ePX/DSC* technology {18}{19}, *ePX Cluster* is directed toward acceleration of complete applications, (i.e. sans application ‘front-end’ and ‘back-end’ components; see *ePX p-code* discussion in reference {19}). In what follows, a theoretical acceleration is derived based upon the *ePX Cluster* template. Non-parallelized process components, (i.e. including multithreaded CPU processes), are assumed pipelined with GPA process components at some specified efficiency. As will be seen from the following simple analysis, CPU/GPA pipelining is critical to the stated goal of accelerating complete applications. In particular, CPU process pipelining is fundamentally dependent upon non-blocking (i.e. ‘asynchronous’) WRITE (‘scatter’) at cluster-node and GPA I/O streams whereby local CPU multithread processing may continue as soon as a work-unit has been written to any process-buffer. In this manner, CPU processes are effectively ‘hidden’ on the global process schedule. At a sufficient degree of pipelining, cluster performance is dominated by component parallel-processes. Qualitatively, we expect any acceleration (‘*A*’) due to parallelization will critically depend upon; (1) the fraction of code than can be parallelized (‘*P*’), (2) the degree of parallelization (‘*N*’), and (3) any overhead associated with parallelization². The basic mathematical model is given by Amdahl’s Law {32}:

$$A = \frac{1}{(1-P) + \frac{P}{N}} \quad (3)$$

This expression is expanded based upon assumed parallel-plus-pipelined code components. In what follows, we assume $C_{CPU} \equiv$ pipeline efficiency constant $\in [0,1]$, $N_{CPU} \equiv$ multicore order, $N_{GPU} \equiv$ GPU array order, $N_{NODE} \equiv$ cluster node order, $P_{CPU} \equiv$ pipelined CPU code fraction, $P_{GPU} \equiv$ GPU accelerated code fraction. A key subtlety is at sufficiently high pipelining efficiency, acceleration terms are restricted to those for which there exists an explicit *scatter-gather*, i.e. $N = N_{GPU} N_{TP/GPU} N_{NODE} = N_{GPA} N_{NODE}$.

$$A = \frac{1}{(1 - C_{CPU} P_{CPU} - P_{GPU}) + \frac{P_{GPU}}{N}} \quad (4)$$

Of particular interest is the limiting case:

$$\lim_{N \rightarrow \infty} \left(\frac{1}{(1 - C_{CPU} P_{CPU} - P_{GPU}) + \frac{P_{GPU}}{N}} \right) = \frac{1}{1 - C_{CPU} P_{CPU} - P_{GPU}} \quad (5)$$

However, with ‘ $C_{CPU} P_{CPU}$ ’ sufficiently large and ‘ N ’ such that:

$$\frac{P_{GPU}}{N} \gg (1 - C_{CPU} P_{CPU} - P_{GPU}) \quad (6)$$

Amdahl's Law then becomes:

$$A_{CLUSTER} \cong \frac{1}{\frac{P_{GPU}}{N}} = \frac{N}{P_{GPU}} \cong N = N_{GPA} N_{NODE} \quad (7)$$

Where the *ePX* supercomputing model is considered, this relation becomes:

$$A_{CLUSTER} \cong N_{GPU} \cdot N_{TP/GPU} \cdot N_{NODE} \quad (8)$$

This result matches the scaling relation cited above, (re: *equation-2*; ' N_{GPU} ' = Number of GPU instances in array, ' $N_{TP/GPU}$ ' = Number of thread processors per GPU), with implication of optimally linear cluster-order performance scaling. However, the condition expressed in *equation-6* also implies, where ' C_{CPU} ' is fixed at some constant value, ' $A_{CLUSTER}$ ' will approach the constant limit expressed in *equation-5*. Stated differently, in absence of perfect CPU/GPA process pipelining, scaling based upon cluster order alone cannot be performed indefinitely. Further, this result can easily be generalized for *any* practical cluster implementation, (i.e. not just *ePX Cluster*). A key point is leveraging of CPU multithreading greatly facilitates optimal pipelining and extension of the linear scaling relation to higher cluster order. In this manner, the cited performance limit may be extended. However, yet another critical subtlety is revealed in form of the essential vectorial nature of any optimal scaling relation for *ePX Cluster*. As previously observed, (re: previous *processing model* discussion), cluster acceleration is constrained by assumption of contention-free internode communication, with implication of a direct variation between component-task size at any cluster-node and cluster order, (i.e. ' $component_task_size(N_{GPA}) \propto N_{NODE}$ ' \rightarrow ' $N_{GPA} \propto N_{NODE}$ '). In this context, *ePX framework* {18}{19} exhibits a most fortuitous property in that ' C_{CPU} ' also varies directly with component task size, (i.e. ' $C_{CPU} \propto component_task_size(N_{GPA})$ '), due to the fact for well designed kernels amenable to SIMD/SIMT processing, the fraction of non-pipelineable code increases only sublinearly relative to associated GPA kernel code. Thus, where *ePX cluster* is considered, the condition for optimal linear scaling, (re: *equation-6*), may in principle be extended to arbitrarily large cluster-order¹⁷.

In practical terms, the assumed PC form-factor will engender limits on both ' N_{GPU} ', (re: limit on motherboard PCIe slot-count), and ' $N_{TP/GPU}$ ', (re: extant GPU technology limitations). Thus, at some scale, the linear-scaling property will become invalid based upon violation of assumed contention-free interprocess communications. At this point, both custom GPU hardware arrays and scalable switch-based network infrastructure may be introduced for extended *ePX* performance scaling.

Summary

ePX Cluster technology is directed toward acceleration of complete applications on GPU-based computational cluster architectures. In this context, adoption of a true supercomputer processing model at all GPU-accelerated cluster nodes facilitates increased CPU/GPU pipelining efficiency cluster-wide. This increased pipelining efficiency can be shown to extend GPU acceleration properties to any cluster-based application amenable to the SIMD/SIMT processing model. This is in contradistinction to acceleration achievable with adoption of any standard CPU/GPU-coprocessor model exhibiting generally inferior CPU/GPU pipelining efficiency. In essence, all application performance-scaling properties of *ePX/DSC* are extended to cluster architectures by virtue of an optimal scaling relation dominated by SIMD/SIMT acceleration at each processing node. *ePX Cluster* achieves this goal based upon a number of innovations:

- (1) Fundamental ‘dynamic-dataflow’ software design representation,
- (2) ‘Distributed-Recursive’ (divide-and-conquer) cluster processing model,
- (3) Multi-Level scatter-gather onto cluster, multicore CPU, and GPA resources based upon hierarchical parsing of dataflow-objects at component-task and algorithmic-kernel granularities,
- (4) Optional datapath virtualization based upon data-server transactions defined at originating cluster-node part and parcel of dynamic-dataflow elaboration,
- (5) Parallelization based upon non-blocking calls to multicore CPU, GPA and cluster resource APIs,
- (6) Flexible application mapping based upon integration of three distinct component processing models; (a) distributed-recursive (cluster), (b) multithreaded RISC (multicore CPU), and (c) SIMD/SIMT (GPU),
- (7) Enhanced software reuse and transportability based upon; (a) adoption of generic process queues for cluster, CPU, and GPA resources, and (b) virtualization of all hardware implementation detail to level of process-queue service routines,
- (8) Enhanced CPU/GPA process pipelining efficiency based upon multicore CPU parallelization/multithreading capability,
- (9) Extended cluster application performance scaling property based upon; (a) increased Amdahl Limit at each cluster-node, (b) vectorial ‘ $N_{NODE} \times N_{GPU} \times N_{TP/GPU}$ ’, (i.e. $N_{NODE} \equiv$ Number Cluster Nodes, $N_{GPU} \equiv$ Number GPUs per Array, $N_{TP/GPU} \equiv$ Number Thread Processors per GPU), scaling property at internode communication bandwidth constraint, (c) sublinear growth of any non-pipelineable code fraction, and (d) use of scalable internode communications infrastructure, (e.g. ‘layered-switch’ network).

ePX Cluster is useful for accelerating any cluster-based software application for which the indicated supercomputing model is appropriate. Broad application categories include; (1) scientific, (2) engineering, (3) mathematical modeling, (4) Bayesian networks, et al. Particular examples include; (1) fluid dynamics, (2) computational weather, (3) computational chemistry/biology, (4) geophysics, (5) astrophysics/cosmology, (6) quantum fields, (7) computational finance, (8) network simulation, (9) solid state physics, et al.

Note¹ – enParallel Xcelleration technology (ePX) - patent pending.

Note² – A ‘precedence’ semantic is imposed with respect to equivalent graph structure; graph ‘node’ is equivalent to some application component possessed of function-binding and directed graph arc implies function at arc-root must complete before function at arc-tip begins.

Note³ – This assertion is based in application of “Rent’s Rule” [26] under conditions whereby all terminals exposed at lower hierarchy elaboration are promoted to top hierarchy, (i.e. cluster internode communications envelope).

Note⁴ – An ‘sg-complete’ (scatter-gather complete) graph is defined as a non-cyclic digraph originating at a single node with outorder > 1 and terminating at a single node with inorder > 1 and for which at least one complete path passes through every node in the graph.

Note⁵ – Interconnect technology standard: point-to-point, bidirectional, serial link with 8b/10b encoding – supports assorted serial rates and channel bonding.

Note⁶ – Ubiquitous LAN technology standard: frame-based, CSMA/CD.

Note⁷ – Analogous to recursive function calls in ‘C/C++’.

Note⁸ – In this context, ‘data’ parallelism refers to a single kernel consisting of more or less identical threads executing in parallel according to the SIMD processing model - ‘task’ parallelism refers to a kernel sequence co-resident in GPU memory, (note this usage of ‘task’ remains distinct from that associated with ‘task’ scatter-gather on cluster infrastructure).

Note⁹ – The scatter-gather abstraction employed by ePX in form of generic operations on process queues greatly simplifies the ePX programming model in that implementation details associated with any specific architectural template or hardware implementation are uniformly pushed to low-level service routines. In this manner, ePX application code remains highly transportable to the level of requiring only build-library customization for any distinct cluster node.

Note¹⁰ – For example, when compared with an alternative approach whereby a single ‘control’ node performs all scheduling and scatter-gather operations, superior parallelization is achieved due to the fact processing on all dataflow branches is not serialized on a single CPU process thread.

Note¹¹ – Distributed scheduler and scatter-gather operations will also distribute all related cluster communications in time resulting in lower internode collision/contention probability.

Note¹² – Important examples include; (1) resolution of dataflow conditionals, (2) structured interprocess communications management, and (3) component parallel process set-up/tear-down.

Note¹³ – At present, ‘conditional scatter-gather’ is restricted to the initiating cluster-node.

Note¹⁴ – In this instance, instructions are already present in GPU workspace, (i.e. ‘device’ memory). Thus, a complete work-unit assembly, and WRITE to the GPA process queue is not required.

Note¹⁵ – In this application component, a large-scale 4-component block diagonal matrix is scattered to concurrent component tasks implementing; (1) linear system solver, (Gauss-Seidel; ‘CT₀’), (2) finite series summation for matrix exponential, (i.e. ‘e^A’; ‘CT₁’), and (3) eigenvalue/eigenvector calculation, (‘QR’ algorithm; ‘CT₂’, ‘CT₃’).

Note¹⁶ – ‘OpenCL’ [33] support will be added when a standard implementation becomes available.

Note¹⁷ – This holds subject to obvious hardware constraints at any given node beyond which locally processed component-tasks are not fully parallelized.

Note¹⁸ – Equivalently, any GPU-accelerated architecture capable of executing ePX Framework software and integrating Cluster, CPU, and GPA processing resources may be substituted for ePX/DSC.

Note¹⁹ – Nominally, 4-10 nodes, depending upon specific hardware configuration.

References

- {1} **“Current Trend of Supercomputer Architecture”** H. Zhang, Univ. Connecticut Dept. of Computer Science and Engineering
- {2} **“GPU Cluster for High-Performance Computing”**, Z. Fan, et al. Center for Visual Computing and Department of Computer Science, ACM/IEEE Supercomputing Conference 2004
- {3} **“SIMT Architecture Delivers Double Precision Teraflops”** W. Wong, *Electronic Design*, #19280 10Jul08
<http://electronicdesign.com/Articles/Index.cfm?AD=1&ArticleID=19280>
- {4} **“NVIDIA CUDA Compute Unified Device Architecture – Reference Manual”**; Version 2.0, June 2008
- {5} **“NVIDIA CUDA Compute Unified Device Architecture – Programming Guide”**; Version 2.0, 6/7/2008
- {6} **“NVIDIA CUDA CUBLAS Library”**; PG-00000-002_V2.0, March 2008
- {7} **“NVIDIA Compute PTX: Parallel Thread Execution”**; ISA Version 1.2, 2008-04-16, SP-03483-001_v1.2
- {8} http://www.nvidia.com/object/tesla_gpu_server.html
- {9} **“GPU Cluster for Scientific Computing and Large-Scale Simulation”** Z. Fan, et al. Stony Brook University ACM Workshop on General Purpose Computing on Graphics Processors 2004
- {10} <http://www.gpgpu.com>
- {11} **“A Performance-Oriented Data Parallel Virtual Machine for GPUs”**; M. Segal, M. Peercy, ATI Technologies, Inc.
- {12} **“Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”**; E. Gabriel, et al. *Proceedings 11th European PVM/MPI Users’ Group Meeting*
<http://www.open-mpi.org>
- {13} **“MPI Parallelization Problems and Solutions”** UCRL-WEB-200945
<https://computing.llnl.gov>
- {14} **“OpenMP Application Program Interface”**; Version 3.0 May 2008
OpenMP Architecture Review Board <http://openmp.org>
- {15} **“MPI: A Message Passing Interface Standard Version 1.3”**; Message Passing Interface Forum, May 30, 2008
- {16} **“MPI: A Message Passing Interface Standard Version 2.1”**; Message Passing Interface Forum, June 23, 2008
- {17} **“Installation and User’s Guide to MPICH, a Portable Implementation of MPI 1.2.7; The ch.nt Device for Workstations and Clusters of Microsoft Windows machines”**; D. Aston, et al. Mathematics and Computer Science Division, Argonne National Laboratory
- {18} **“GPU-based Desktop Supercomputing”**; J. Glenn-Anderson, *enParallel, Inc.* 10/2008
- {19} **“ePX Supercomputing Technology”**; J. Glenn-Anderson, *enParallel, Inc.* 11/2008
- {20} Beowulf Cluster: <http://www.beowulf.org>
- {21} Sun Grid Engine: <http://gridengine.sunsource.net>
- {22} **“Beginner’s Guide to SUN GRID ENGINE 6.2 Installation and Configuration”**
<http://www.sun.com> White Paper September 2008
- {23} **“The MOSIX Distributed Operating System: Load Balancing for UNIX”**; A. Barak, et. al. Springer-Verlag *Lecture Notes in Computer Science* June 1993
- {24} **“The MOSIX Multicomputer Operating System for High Performance Cluster Computing”**; A. Barak, O. La’adan *Institute of Computer Science Hebrew University of Jerusalem*

- {25} ***“The MOSIX2 Management System for Linux Clusters and Multi-Cluster Organizational Grids”***; A. Barak A. Shiloh <http://www.MOSIX.org>
- {26} See http://en.wikipedia.org/wiki/Rent's_Rule
- {27} ***“InfiniBand Architecture Specification Volume 1 Release 1.2.1”***; InfiniBand Trade Association November 2007 Final Release
- {28} ***“1000BASE-T Gigabit Ethernet Tutorial”***; Hewlett-Packard Company September 15,2000
- {29} ***“CUDA Version 2.1”*** download: http://www.nvidia.com/object/cuda_get.html
- {30} ***“Symmetric Multi-Processing (SMP) Systems on Top of Contemporary Intel Appliances”***; Jiri Hlusi, Thesis – University of Tampere, Dept. Of Computer and Information Sciences December 2002
- {31} ***“TCP/IP Tutorial and Technical Overview – Understanding Networking Fundamentals of the TCP/IP Protocol Suite”***; L. Parziale, et al. IBM International Technical Support Organization December 2006
- {32} ***“Principles of Parallel Programming”***; C. Lin, L. Snyder 1st Ed. Addison-Wesley 2008
- {33} ***“The OpenCL Specification”***; Khronos OpenCL Working Group, A. Munshi Ed. Version 1.0, Document Revision 29