Critical Technology Evaluations

Technology	Description	Status
Runtime Scheduling	Priority-queue scheduling for infinite level priorities	Evaluating
Data Placement	Communication avoiding data placement for Cholesky Decomposition	Evaluating
Runtime Data Migration	Prefetch mechanisms for pull-based data migration	Identified
Runtime Self-Awareness	Metrics for modulating prefetch depth	Identified
Compiler	Improving hierarchical mapping capabilities in R-Stream	Identified
Compiler	R-Stream SWARM backend and runtime layer	Identified
Parallel Libraries	HTA library implementation in PIL	Evaluating
Parallel Libraries	Extending PIL to include libraries	Evaluating
Applications	Identified two NWChem kernels for study	Complete
Applications	Creating C-only version of the Self-Consistent Field computation	In progress

Summaries of Quarterly Work

Improving hierarchical mapping capabilities in R-Stream

In the context of SWARM, hierarchical mapping is the problem of creating a hierarchy of codelet graphs to be scheduled by SWARM. The different levels of the hierarchy may run code on different platforms, including x86 nodes and clusters and CUDA. While some success has been met in the context of producing hierarchically mapped programs on multiple GPGPU devices using CUDA, there is a scalability problem related to the need to represent inner loops of the hierarchy as high-dimensional objects. We are working to reduce the dimensionality of these objects at each level of the hierarchy.

R-Stream SWARM backend and runtime layer

The Reservoir Labs team studied both the SWARM and SCALE forms and defined that SWARM would be a more straightforward target, for its closer resemblance to C and the fact that more aspects can be tuned (LIFO vs FIFO scheduling) through SWARM at the moment. The initial approach will be to dynamically declare a task graph. Each task of the graph will be associated with one input dependence which will be satisfied when all the predecessor tasks have validated it. The graph declaration will basically count the number of predecessors for each graph before creating the dependence. An extra parameter will be passed to each task that defines the set of its successor dependences. A function that satisfies all these dependences will be called at the end of the task.
The initial version will target a one-node system, and will hence include SWARM’s network functions.

Cholesky Decomposition as a testbed for Scheduling, Memory Management, and Self-Awareness

As the co-design apps are being developed, we chose to use existing applications to test algorithms in scheduling, data placement, data migration, and self-awareness. We have found that a finite-priority level queue is insufficient for this application and have developed a priority scheduler to address this. Further, we have determined optimal ways of laying out the data to minimize communication between the nodes while balancing work. Finally, we found that it was necessary to observe metrics of outstanding work and outstanding data requests and build a model to modulate the request of additional data when these dynamic parameters vary from their setpoints. Details are listed below in the “Cholesky Decomposition” section of “Topic Detail”. We intend to publish a paper next quarter describing our performance results and findings.

NWChem Kernel extraction

We have studied the NWChem code and identified two kernels for study under the DOE XStack program:

- Self-Consistent Field Computation
- Coupled Cluster Method

We have extracted both kernels and are developing clean, self-contained, C-only versions of both kernels. We are preparing input and output files for testing. Details can be found below in the "NWChem Kernel" section of "Topic Detail".

Parallel Libraries

The Parallel Intermediate Language (PIL) has been extended to allow library functions to be represented as collections of PIL nodes. PIL programs at the beginning of the project could only be a collection of nodes linked by dependence arcs that represent the parallel execution of a program in terms of codelets. With the new extension, one can now implement a subgraph that performs a particular operation and include it as a library function. This works by making it possible to enter the subgraph at any point in the execution of the program, suspend the execution of the program, perform the desired operation by executing the codelets in the order enforced by the dependences, and at the end of the process return execution control to the program that made the request. We are now working on our first design and implementation of the Hierarchically Tiled Arrays (HTAs) library using the new PIL library feature.

Topic Detail:

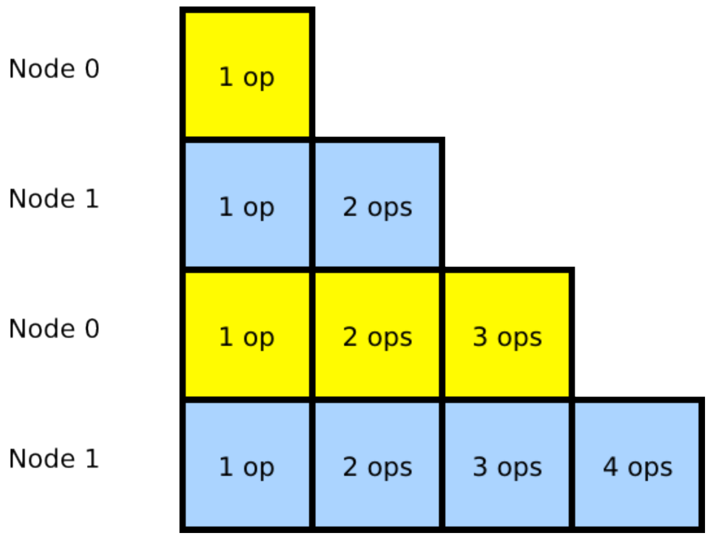
Cholesky Decomposition

The clustered implementation of Cholesky decomposition application presents us with unique challenges for prioritization, work balance and memory management. Cholesky decomposition is a matrix operation, and we solve it by breaking the matrix into tiles and modeling the dependencies between tiles as a DAG; see this paper for more information on the problem and the tiled approach to solving it.

Data Distribution & Work Imbalance

In our implementation, compute nodes are assigned full rows of tiles, and every node typically has many rows.

One issue with this is that of work imbalance. It is advantageous to keep the nodes making progress at roughly the same rate, for several reasons. The first and most obvious reason is that if one node has more (total) work to do, then, at the end of the job, all of the other nodes end up waiting for it, which wastes CPU cycles and increases the total execution time. When rows of tiles are assigned to nodes in a simple round-robin fashion, workload imbalance exists. Here is a trivial example with 2 nodes and 2 rows per node:

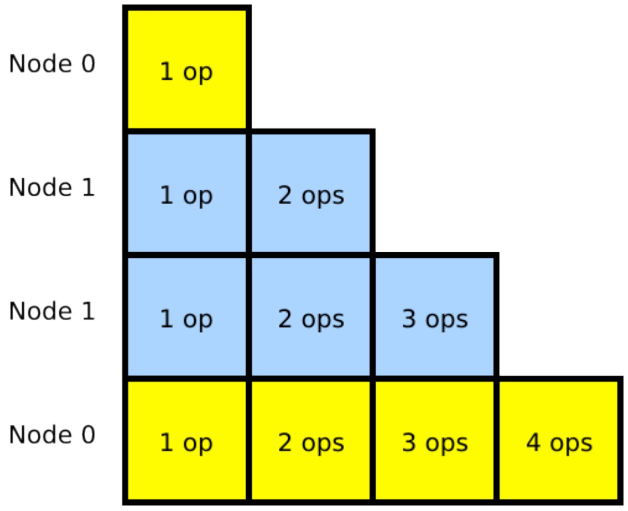


Node 0 owns the yellow tiles, and node 1 owns the blue tiles. As you can see, node 0 owns 4 tiles, and is responsible for doing 7 tile operations. Node 1 owns 6 tiles, and is responsible for doing 13 tile operations.

The work imbalance problem persists as the node/row count increases. For example, if you have 64 compute nodes and each node has 14 rows of tile data, node 0 has 5838 tiles, and must perform 1686062 operations on those tiles. Node 63 has 6720 tiles, and must perform 2082080 operations on those tiles. Thus, node 63 has 15% more tiles than node 0, and must do 23% more work than node 0. Since node 63 has the most work to do, node 63 is the bottleneck for this application.

Obviously, the simplest approach is to assign rows to nodes in a round-robin fashion. However, as suggested above, this leads to poor workload distribution. To resolve this, we have tried some alternative row assignment strategies.

First, we tried a simple reversal, such that row assignment progresses in round-robin fashion for most of the rows, and then at some point, the assignment flips, so rows are assigned in reverse round-robin fashion. If the assignment flips at the right point, the amount of work evens out.



Again, node 0 owns the yellow tiles, and node 1 owns the blue tiles. As you can see, each node now owns 5 tiles. Node 0 is responsible for performing 11 tile operations, and node 1 is responsible for 9 tile operations. The difference in workload is now 11-9, rather than the 13-7 it was previously, so this is an improvement. With a sufficiently large number of rows per node, it is usually possible to find an inversion point which results in less than 1% workload imbalance.

This solves the problem of overall workload imbalance. However, there is also a problem of internal workload imbalance. Given a sufficiently large number of tile rows, data must be passed between nodes all the time in order to keep the nodes busy. (There will be more discussion of this in the Task Prioritization section, below.) If one node is progressing at a slower rate than another node, then that node will not produce the data necessary for the other node to continue progressing.

To improve the internal workload balance, we modeled a more complicated reversal pattern, where the node assignment occurred in forward order, then reverse, and then repeats, going forward, then reverse again, etc. This results in fairly even distribution for a sufficiently large number of rows per node. For instance, for 64 nodes and 14 rows per node, the workload imbalance is 1.5%. An even more complicated pattern, consisting of forward, reverse, reverse, forward, is even more balanced than this. For 64 nodes and 14 rows per node, this pattern has a workload imbalance of 0.5%. This is the pattern we are currently using.

Scheduling: Task Prioritization

Computing a tile operation in this matrix usually requires data from previous tiles. Depending on the operation, the required tile data is always to the left in the same row, or above in the same column, or both left and above. Because entire rows of tiles live on a node, intra-row dependencies are always satisfied locally. These intra-row dependencies are quite common. The data can also be satisfied locally when the data is on another row, but that row is also owned by the same node. When a dependency cannot be satisfied locally, a copy of the data is sent from the node which produced it, to all nodes which need it. On a node receiving this data, a temporary buffer is allocated to store it. The node keeps track of how many tiles require this data, using a counter. When the counter drops to 0, the memory is freed.

When running on multiple compute nodes, just keeping the work queue full on one node is no guarantee other compute nodes will also be kept busy. Since each node depends on data from the others, it is important that work is done in an advantageous order, so that such tasks are finished in a timely fashion. Assigning priority levels to types of tasks (i.e. POTRF, TRSM, GEMM) is not sufficient for Cholesky, because most high-priority tasks depend (eventually) on a lower priority task, and it is not clear that priority inheritance systems would help. We found that assigning priority based on a tile's location within the matrix and the phase of the computation is advantageous, as this ensures that parallelism is exposed as early as possible. This solves the problem of making sure other nodes are kept busy, essentially by making sure we compute the tasks depended upon by other nodes, and send the resulting data to those nodes, as quickly as possible. We also found that this helps to keep a single node busy when that node has a huge thread count, such as Intel Xeon Phi.

SWARM's priority system, as it exists today, has 4 priority tiers. This does not allow sufficient granularity to implement priority based on matrix location, except for very small matrices.

Such a system would need at least 1000 priority tiers in order to be useful for this task. To accommodate a much larger priority space, we added a separate work queue to the application. This work queue is an AVL tree which is sorted by the tile's position within the matrix. Worker threads find the "least" value in the AVL tree, dequeue that and work on it.

We found that multiple sort orders were effective for this tree. We can bias toward the top-most row, by sorting on Y first, and then X. We can also bias toward the left-most column, by sorting on X first, and then Y. For now, we settled on X-first sorting, for memory usage reasons.

Memory Usage for Intermediate Data

As mentioned above, remote data is copied around in order to provide input to local tasks. The priority system now tries to send these copies as quickly as possible. These copies take up memory, and as the matrix size increases, the memory consumed by these copies can exceed the total amount of system memory. This becomes a big problem on 32 nodes and above.

Solving the memory problem requires two things. First, the work must occur in an order that focuses on freeing the network buffers as quickly as possible. Second, care must be taken to avoid sending too much data to a node at a time.

Both types of network data (POTRF results and TRSM results) are consumed by a vertical column of tiles below them. This means, an execution pattern which goes column by column (as is the case with X-first sorting) will do all of the work necessary to free up a network buffer, and then the work necessary to free up another network buffer, etc. If a node is currently working on one column, and will work on the column to the right after this, it is easy to calculate which input data will be needed, and request that data shortly before it is needed. If data is pre-fetched in this fashion, rather than sent automatically when it becomes available, it is now possible to bound the amount of memory used by network buffers at any given time.

So, we added a request/response handshake so that a node can request the TRSM data it needs before it needs it. (There are vastly more TRSM buffers than POTRF buffers, so our solution focuses only on the TRSMs.) We combined this with some simple logic to determine how "far along" the local node is in the overall matrix calculation, we can project these requests several tile-columns out in front of the current computation point, as a form of network prefetch. If all goes well, the data arrives just before it is needed, and is then consumed, resulting in the minimum memory usage.

Unfortunately, not all columns are created equal. The memory footprint of a column's inputs varies throughout the course of execution; it starts out small, for low-numbered columns, and gets quite large for high-numbered columns. The following table shows an example of this effect. It describes the prefetch memory usage in a 64-node job, with 14 rows per node, for a total of 896 rows of tiles in the matrix, when attempting to prefetch 10 columns ahead of the current execution point.

Current column #	Inputs on-hand	Memory usage

5	5..15	110 tile copies
20	20..30	275 tile copies
400	400..410	4455 tile copies
700	700..710	7755 tile copies

So, this system quickly becomes a balancing act. We want to expose enough parallelism to keep the system busy, for low column numbers, without running out of memory once the program progresses to a higher column number. Performance varies widely with this prefetch threshold setting, and careful tuning is required.

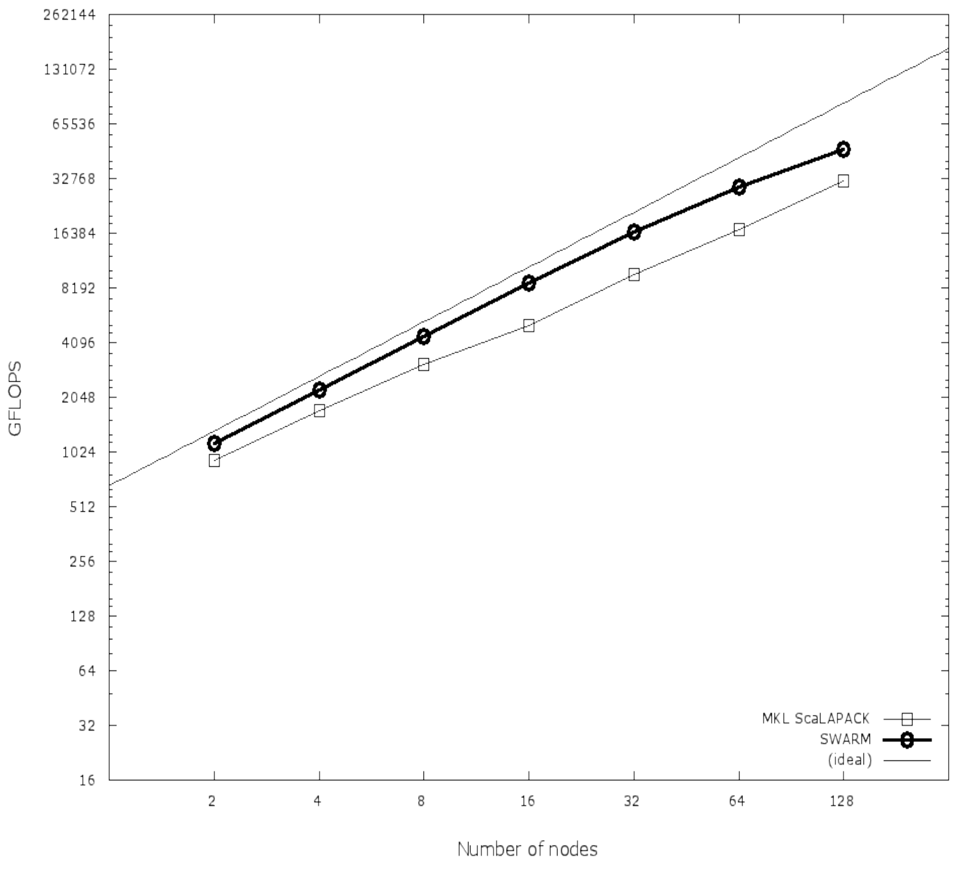
Self-Awareness and Control

The next step is to find a more robust autotuning solution to this problem, that measures dynamic runtime parameters and controls the prefetch request window to achieve the appropriate set point of memory consumption and concurrency.

To achieve this, we maintain a low-water mark of outstanding requests. The application maintains two counters, one count of how many tile copies have been requested, and another count of how many tile copies have actually arrived. The request count is the potential max memory usage; the copy counter is the current memory usage. The user supplies a number of tile copies on the command line, which is used as the goal. When a request is sent, the request count is increased by the number of expected packets. When a packet arrives, the copy count is incremented. When a packet is freed, both counts are decremented. Whenever the current number of requested tiles dips below this threshold, another request is kicked off. Once the last column in the matrix has been requested, the prefetch system disables itself and the request count is allowed to drift slowly toward zero.

On Endeavour, the compute nodes have 64GB of memory. We scale the matrix size with the node count, in order to keep the per-node data size close to constant. For this system, we found a low-water mark setting of 40000 works well; this equates to about 25GB of network buffers per node.

Results



NWChem Kernel

We have selected the Self-Consistent Field (SCF) calculation and Coupled Cluster Method (CCM) as the first two NWChem Kernels to provide the DOE XStack teams. Both are stand alone application benchmarks distributed with the Global Array software distribution. The distributed versions use Global Arrays, include both C and Fortran source code files, and call standard math library routines. To facilitate the port of the kernels to the Exascale execution models being developed for XStack, we are producing clean, self-contained, C-only versions.

SCF is an iterative, fixed-point process for solving the electronic Schrodinger equation of molecular systems. A Fock Matrix is constructed from one- and two-electron integrals and solved. The two-electron integral process is the computationally dominate process as it scales as n^4 in the number of basis functions. Its structure is idiomatic of other key kernels in computational chemistry codes, including Density Functional Theory and the Coupled Cluster Method. SCF iterates until certain desired numerical tolerances are achieved or a maximum number of iterations is reached.

We started with the GA stand alone application that uses the basis sets for Beryllium atoms. The system is of moderate size taking a few minutes to solve on an Intel X86 manycore system. Removing the GA code has been straightforward. The GA version creates a queue of tasks for each parallel process. Tasks are associated with tiles of one or more of the application's principal two-dimensional data structures: the Schwarz, density, and Fock matrix. The structure of each process is something like …

```
while there are tasks on the queue {
  get next task()
  get tiles for task (iLo, iHi, jLo, jHi)

  for i = iLo to iHi {
  for j = jLo to jHi {
      … process code …
} } }
```

To produce a C-only version of each process, we removed the task loop, the GA calls, and changed the ranges of the loops to dimensions of the data structures processed, i.e,

```
for i = 0 to nbfn - 1 {
for j = 0 to nbfn - 1 {
      … process code …
} } }
```

where *nbfn* is the number of basis functions.

The GA version calls Fortran modules for matrix multiply and finding eigenvalues. We are rewriting those routines in C to provide a single language kernel.