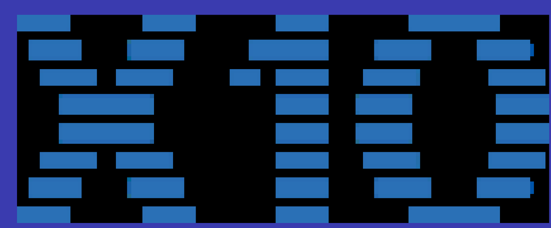


X10 as a parallel language for scientific computation

We are exploring the suitability of the X10 programming language for development of high performance scientific applications.

We developed three different scientific codes entirely in X10 version 2.1. One performs a complete **Hartree-Fock** (HF) quantum chemistry computation on a molecular system. The other two implement alternative approaches for the fast calculation of long-range electrostatic forces in biochemical simulations: the **Fast Multipole Method** and the **Smooth Particle Mesh Ewald method**. Each application represents a different pattern of computation & communication.

Our feedback on the applications presented here led to improvements in the X10 implementation including: **complex** arithmetic, fast local **arrays**, scaling of **distributed arrays** and **async**.

Hartree-Fock method

The Hartree-Fock method (HF) is widely used in quantum chemistry to determine the motion of electrons around fixed nuclei.

The core of the method is the creation of the **Fock matrix**:

$$F_{ij} = H_{ij}^{core} + \sum_{k,l}^{N} P_{kl}[\langle ij|kl\rangle - \frac{1}{2}\langle ik|jl\rangle]$$

of which the most expensive part is the evaluation of four-centered **two electron integrals** $\langle ik|jl\rangle$.

Overall, this problem is characterized by **load imbalance** in the evaluation of the two-electron integrals and the need to **accumulate** multiple contributions to every element in the Fock matrix.

We tested a number of load balancing schemes suggested by Shet et al.[1]. Dynamic load-balancing schemes were not efficient due to high overheads associated with **async** activities and **atomic** sections. The best scalability was achieved with static, program-managed load balancing as follows:

```
// setup
val computeInst =
 DistArray.make[ComputePlace]( Dist.makeUnique(),
   (Point) => new ComputePlace(N, molecule, basisName)
);

// each iteration - copy updated density matrix to each place,
// compute local F contribution
finish ateach (p in computeInst) {
 val comp_loc = computeInst(p);
 comp_loc.reset(density);
 comp_loc.computeShells(nPairs);
}
// [gather and reduce F matrix]
```

Application codes and test suites available at
http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html

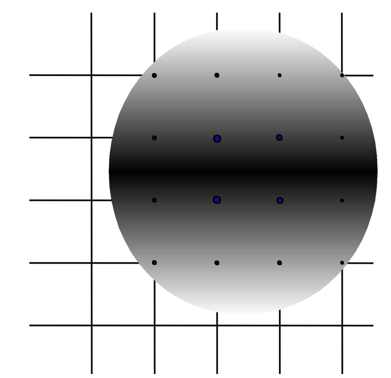
References

[1] A. G. Shet, W. R. Elwasif, R. J. Harrison, and D. E. Bernholdt, *"Programmability of the HPCS languages: A case study with a quantum chemistry kernel"*, in Proceedings of IPDPS '08, IEEE (2008).
[2] U. Essmann, L. Perera, M. Berkowitz, T. Darden, H. Lee, and L. Pedersen, *"A smooth particle mesh Ewald method"*, J. Chem. Phys. 103 (1995).
[3] L. Greengard and V. Rokhlin, *"A fast algorithm for particle simulations"*, J. Comp. Phys. 73 (1987).

Josh Milthorpe & Alistair Rendell
Computer Systems Group
College of Engineering & Computer Science

Particle Mesh Ewald method

The Smooth Particle Mesh Ewald method[2] splits particle interactions into short-range and long-range components. The **short-range** component is summed within some reduced domain. The **long-range** component is calculated by approximating a charge density field and interpolating charges on a **mesh** of grid points. Complexity is **O(N log N)**.



A key step is a **3D Fast Fourier Transform**, which requires **all-to-all** communication for a distributed implementation.

Our code uses **active messages** and **global barriers** to implement the all-to-all communication. A more efficient implementation would use a blocking collective operation, similar to **MPI_Alltoall**, to combine data transfer and synchronization.

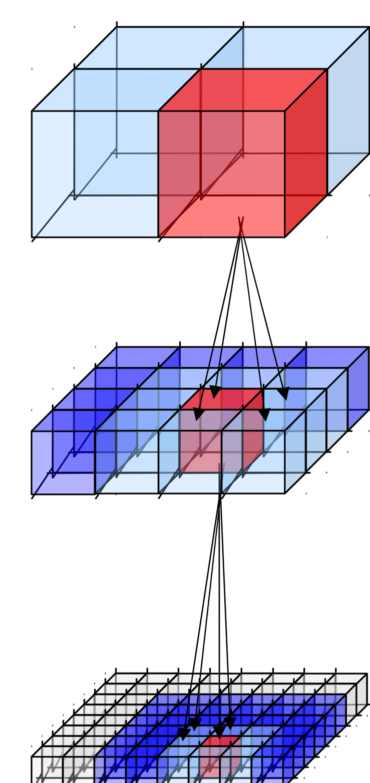
The all-to-all transpose is implemented in X10 as:

```
def transpose(
 source:
  DistArray[Complex](3),
 target:
  DistArray[Complex](3)) {
 finish {
  for (p2 in
     source.dist.places()) {
    val transferRegion:
     Region(3) = //
    val toTransfer = // ...
    async at (p2) {
     var i: Int = 0;
     for ([x,y,z] in
       transferRegion) {
      // transpose dimensions
      target(z,x,y) =
        toTransfer(i++);
} } } } }
```

Fast multipole method

Fast Multipole methods[3] are also used to evaluate electrostatic interactions in molecular dynamics, with complexity of **O(N)**.
The simulation space is divided into an **octree** of cubic boxes. Interactions between particles in nearby boxes are evaluated directly, whereas distant interactions are evaluated using **multipole expansions** around box centers.

The distributed FMM is characterized by complex data structures and **localized communication** patterns.
Steps in the algorithm:
- generate multipole expansions at lowest level (P2M);
- upward pass, combining expansions at higher levels in the tree (M2M);
- transform to local expansions (M2L);
- downward pass, translate local expansions to lower levels (L2L);
- evaluate far-field interactions using local expansions (L2P);
- directly evaluate near-field interactions (P2P).



The X10 struct type **GlobalRef** is a reference to an object at one place that may be passed to other places. GlobalRef can be used to build arbitrary **distributed data structures**. In the FMM code, it is used in the octree to store pointers to parent boxes held at a different place.

The following code retrieves the local expansion for the parent box, used in the L2L step:

```
// at parent place
val parent =
 GlobalRef[FmmBox]
   (parentBox);
// at child place
val parentExpansion =
 at (parent)
 {parent().localExp};
```

In MPI, this would be done through a pair of **MPI_Send** and **MPI_Recv**, synchronizing source and destination processes.

The X10 class libraries provide a number of commonly used **distributions** including **block**, **cyclic** and **unique**. Users may create custom subclasses of x10.array.Dist to support more complex geometric decomposition. Our FMM code includes a custom **Morton distribution** for the octree.