CPE: A Parallel Library for Financial Engineering Applications

The Clustertech parallel environment is an object-oriented C++ library that uses abstractions to simplify parallel programming for financial engineering applications. The message passing interface ensures CPE's portability and performance over a wide range of parallel cluster and symmetric multiprocessing machines.

Monk-Ping Leong
Chi-Chiu Cheung
Chin-Wang Cheung
Polly P.M. Wan
Ivan K.H. Leung
Winnie M.M. Yeung
Wing-Seung Yuen
Kenneth S.K. Chow
Cluster Technology Ltd.

Kwong-Sak Leung
Philip H.W. Leong
The Chinese University of Hong Kong

Parallel computing has emerged as a cost-effective means of dealing with computationally intensive financial and scientific problems. To effectively utilize this technology, developers need software that reduces the complexity of the process as well as tools to support integration of parallel and desktop machines.

The Clustertech parallel environment (CPE) is a C++ library that facilitates development of large-scale parallel applications, particularly financial engineering applications. Written with performance and portability in mind, CPE currently runs on the Unix, Linux, and Windows operating systems.

CPE provides domain-specific object-oriented libraries for solving partial/stochastic differential equations using the finite-difference method and Monte Carlo simulation. These libraries factor out the common operations required for FD and MC computations so that in most cases the user need only provide the code required for the specific application.

CPE hides parallel synchronization and communications, allowing the user to emulate conventional serial programming; it also offers users better control of parallelization by overriding default methods. The domain-specific libraries are built on top of a set of high-performance parallel programming classes that ensure efficient communications and control.

Although researchers have developed specialized parallel libraries for solving partial differential equations,[1] we are unaware of any other object-oriented parallel libraries for financial engineering applications that offer CPE's features.

Sophisticated users may elect to execute programs directly on the parallel platform, but most commercial applications require integration of the parallel routines with existing software on the user's desktop machine—for example, an Excel spreadsheet, a Web-based interface, or a custom program. CPE provides mechanisms to seamlessly call and control parallel computations remotely from a desktop machine and transfer data within and between parallel machines and the desktop.

CPE introduces several abstractions to simplify parallel application development. At the lowest level, a Tx class and related drivers unify communications, easing the task of transporting complex objects over different protocols. The MC implementation employs policies[2] to provide flexible control of the execution of these simulations. A *distributed grid* together with *expression templates* facilitate the implementation of partial differential equation solvers, allowing efficient manipulation of entire parallel grids using simple operators. Finally, *remote execution* enables the creation and manipulation of parallel objects from a desktop machine.

CPE ARCHITECTURE

As much as possible, CPE decouples parallelization from the problem description. The libraries support both serial and parallel execution from the same source code so that in most cases users can debug a serial version of the program before testing the same code as a parallel program. Debugging is simpler because the serial versions run within a single process.

CPE uses C++ templates extensively to support arbitrary datatypes and to implement a policy mechanism. It also employs expression templates[3] where possible to minimize unnecessary object creation. To simplify the code required for transferring user-defined complex datatypes, CPE relies on a metacompiler to generate ANSI C++ code, which is compiled using a standard compiler.

The CPE architecture consists of three layers:

- The *data abstraction and transportation core layer* provides the Tx class and transportation drivers that facilitate communications among parallel processes, desktop machines, and databases via the message passing interface (MPI), Extensible Markup Language (XML), base64 encoding, and open database connectivity (ODBC). This layer simplifies the code associated with transferring complex data among different types of processes and machines.
- The *parallel application layer* provides the domain-specific FD and MC libraries.
- The *remote execution layer* provides an interface between the parallel computation and the desktop machine. This layer introduces remote parallel objects, which can encapsulate applications built using the domain-specific libraries. Users can manipulate the handles of these parallel objects on a desktop machine to access applications residing on the parallel platform.

The core layer uses standard MPI primitives for communications and synchronization to ensure portability and high performance. Developers can either write applications to be launched from a shell via the standard MPI mechanism or organize them as remote parallel objects to be launched by CPE's remote execution module. In addition, the domain-specific libraries and remote execution layer transfer and marshal complex data objects via this layer. Developers can also use this layer to construct other applications outside these domains.

DATA OBJECT TRANSFER

Much of the code in a parallel program deals with transferring data between coprocesses. Libraries such as MPI do not directly support the transfer of nonprimitive objects like Standard Template Library (STL) containers and user-defined structures. Manually manipulating data inside all data structures places a heavy burden on developers.

CPE uses C++ templates extensively to support arbitrary datatypes and to implement a policy mechanism.

The CPE data abstraction and transportation layer offers a unified mechanism for transferring objects via the Tx class; it provides a similar ConstTx class for read-only objects.

Tx's task is to describe the decomposition of data objects down to primitive types. Tx transportation drivers use existing libraries such as MPI and libxml2 for data transfer among parallel processes and marshaling to XML. They use Tx's description to automate MPI transportation and marshaling.

Tx supports abstraction of STL containers—including vector, deque, list, set, map, and string—as well as nested combinations of such containers. To take an extreme example, CPE can send a vector< map< string, list<double> > > via MPI in one statement, as easily as sending a double. To support this feature, CPE uses compile-time template metaprogramming.

CPE automatically generates code for describing fields in a user-defined struct (or class) and template struct with a metacompiler developed using OpenC++.[4] The CpeTx class modifier identifies the classes that require such code generation. As an example, for

```
CpeTx class Bermudan
{
public:
    double initialPrice;
    std::vector<double> exerciseTimes;
    std::vector<double> strikePrices;
};
```

CPE generates a list of the member variable types—a double and two vectors of doubles—and methods to obtain the names and offsets of the member variables—namely, initialPrice, exerciseTimes, and strikePrices. CPE also supports members that are objects, multiple inheritance, and virtual inheritance.

Using the Tx class, CPE can convert an object b1 of the Bermudan class to a document object model (DOM) tree using the XML driver as follows:

```
Xml::toDomTree(tree, Tx(b1));
```

The availability of implicit constructors makes it possible to omit the Tx() call.

The MC library in CPE facilitates the parallelization of MC applications.

The same structure can be broadcast via MPI using

```
Communicator::world().broadcast(b1);
```

Because MPI supports only fixed-length transfers, this broadcast must occur in two stages: CPE transfers initialPrice and the two vectors' lengths in the first stage and the vectors' data in the second. It applies template metaprogramming recursively at compile time to automatically determine the optimal schedule for such transfers.

To support the transfer of subsequences of sequence containers, CPE introduces data access patterns via the ContiguousPattern, VectorPattern, and IndexedPattern classes. Specifying a pattern to a Tx object's constructor allows the transfer of selected elements in the containers. For example, in the code fragment

```
std::vector<double> vec(15);
Tx tx(vec, VectorPattern(3, 2, 5));
```

the VectorPattern constructor's first argument is the number of blocks the pattern contains, the second is the number of elements in a block, and the last is the number of elements between neighboring blocks, or the *stride*. The Tx object refers to three blocks of elements, each block consists of two elements, and successive blocks are five elements apart. Therefore, the pattern represents the elements of vec with indexes 0, 1, 5, 6, 10, and 11.

MONTE CARLO SIMULATION

MC simulation is a numerical technique for solving problems that stochastic models describe by generating numerous samples, commonly known as *paths*. Computation speed is a major barrier to deploying MC simulations in many large and real-time applications. The MC library in CPE facilitates the parallelization of MC applications, freeing users from dealing with these issues while maintaining extensibility. The implementation works on heterogeneous clusters and uses dynamic load balancing.

MC library features

The MC library

- supports conventional MC simulations (using pseudorandom numbers), quasi-MC simulations (using low-discrepancy sequences),[5] and simulations based on a set of predefined scenarios;
- supports collection, consolidation, and reporting of cross-sample intermediate results during the simulation;
- supports complex termination criteria with compound logical and/or relations; and
- stores samples in memory for future reuse.

The MC library provides the Simulation abstract base class and related classes that encapsulate a parallel simulation. To use the library, the user simply codes the computation of a single sample via an abstract method. The MC library can then handle most other aspects associated with the parallelization and load balancing of the MC simulation in a manner transparent to the programmer.

Example

As a derivative pricing example, consider a two-asset model with two lognormally distributed stocks that evolve under the Black-Scholes stochastic differential equation (SDE):[6]

$$dS_1 = \mu_1 S_1 dt + \sigma_1 S_1 dZ_1$$
$$dS_2 = \mu_2 S_2 dt + \sigma_2 S_2 dZ_2$$
$$E[dZ_1 dZ_2] = \rho_{12} dt$$

where μ, σ, and ρ are the drifts, volatility, and correlation coefficient of the stocks, respectively.

The payoff of a European exchange option that allows the owner to exchange one unit of S_2 to S_1 at maturity is $max(S_1 - S_2, 0)$. The sample generation code can be derived from the SDE using the Euler scheme:

```
bool computeOneSample(double& sample)
{
    ... /* draw dZ from RNG uniquely seeded in
    each co-process */
    for (t = 0; t < endT; t += dt) {
        s1 += mu1 * s1 * dt + sigma1 * s1 * dZ[0];
        s2 += mu2 * s2 * dt + sigma2 * s2 * dZ[1];
    }
    /* Compute option price at maturity. */
    sample = (s1 > s2) ? (s1 – s2) : 0.0;
    ...
}
```

The MC library parallelizes the simulation by controlling each coprocess to run sample generation, with intermittent communications to schedule, coordinate, and load-balance the remaining simulation. The Simulation class run method performs the parallel MC simulation in a series of computing and synchronization steps.

In the computing step, the MC library calls computeOneSample repeatedly and appends each output sample to a localSamples list, a member variable in each parallel coprocess. During synchronization, the library coordinates coprocesses and consolidates samples to produce statistics as well as handles progress checking, reporting, and, optionally, user-defined actions. Users can keep the samples in localSamples or remove them after consolidation to release memory.

After synchronization, the MC library checks the termination criterion to determine whether to terminate or start another computing step. The MC library is optimized to minimize both idle processor cycles and communications overhead.

Policies

Although MC simulations have much in common, different applications require various methods to perform sampling, consolidate statistics, and terminate the simulation. The MC library implements these as pluggable policies, which are template parameter types of the host class. The policies include

- *sampling*—for sample computation using different kinds of inputs or methods;
- *consolidation*—for consolidating samples to statistics; and
- *termination*—for specifying the termination criterion.

The MC library provides predefined policies to cover common tasks. For example, a Simulation class with double-precision floating-point sample data and random sampling (no input argument to the method required for computing a sample) that consolidates mean and standard error statistics and terminates when the standard error reaches a predefined target is declared as Simulation <double, Random, TrackMeanAndStdErr, TargetStdErr>.

In the European exchange option example, the user's program can obtain the price (mean of the samples) and its accuracy (standard error of the mean) directly from the consolidation policy.

A special consolidation template policy, AndThen, lets users cascade consolidation policies. Two special termination template policies, And and Or, allow users to combine termination criteria with logical relations. These policies can be nested to an arbitrary number of levels.

The MC library also supports an Indexed sampling policy that calls the user-defined sample computation method bool computeIndexedSample (IndexType index, double& sample) *N* times, each time with a unique index value [0, 1, ..., *N* – 1]. To avoid unnecessary synchronization, multiple coprocesses can handle the same index, in which case the MC library removes duplicates during synchronization.

The MC library provides predefined policies to cover common tasks.

The user's program can use this policy to implement quasi-MC simulations[5] that replace the random numbers in MC simulations with a low-discrepancy sequence and use each *lattice point* in the sequence exactly once.

In the European exchange option example, the class that uses a multidimensional Niederreiter sequence[5] is Simulation<double, Indexed, TrackMeanAndStdDev, CompleteAllIndices>. The sample computation method takes index as input and uses it to obtain the lattice point in the Niederreiter sequence:

```
bool computeIndexedSample(IndexType index,
double& sample)
{
    std::vector<double> niedPoint;
    seqGen.getElementAt(index, niedPoint);
    //Use niedPoint to generate dZ and calculate
    option price.
    ...
}
```

The user's program obtains the *i*-th lattice point (a vector) by calling the getElementAt method of the seqGen sequence generator object. The CompleteAllIndices policy terminates the simulation after it has used all the indices.

The MC library provides another sampling policy, Parameterized<ParamType>, that is a template itself in which ParamType is the datatype of parameters for computing samples. The user must supply an array of sample parameters, denoted as [P_0, P_1, ..., P_{N-1}] (each P is of the ParamType datatype), to the simulation object. The method bool computeParameterizedSample(const ParamType& param, SampleType& sample) takes a ParamType parameter as input and performs the sample computation.

This sampling policy is similar to the Indexed sampling policy except that it uses a user-defined type parameter (P_{index}) rather than the integral value index as input to the sample computation method. It is especially useful in value-at-risk calculations in historical simulations, in which ParamType is a structure containing parameters describing the historical scenarios.

FINITE-DIFFERENCE CALCULATION

Solving a partial differential equation requires

CPE's FD library provides a multidimensional distributed array for representing grids.

formulating a *difference equation*—the updated value of each grid point as a function of the grid point and its neighbors at the previous time step. CPE's FD library provides a multidimensional distributed array for representing grids. The library spatially decomposes a grid into domains and binds each to a parallel coprocess. The number of grid points that a coprocess must handle decreases as parallelism increases.

FD library features

The FD library centers on a Grid class. To facilitate applying the difference equation on grid points at a domain decomposition's boundaries, users can specify *ghostzones* of an arbitrary width for a Grid-class object. A ghostzone is the extension of a domain in which the grid point values are obtained from neighboring domains. Ghostzone synchronization, the process of updating the values of grid points in ghostzones from other coprocesses, occurs using the sync method.

The FD library's automatic domain decomposition minimizes communications between coprocesses. The library applies a recursive algorithm to enumerate all the ways a grid can be divided into subgrids and then chooses the option with the fewest grid points in the ghostzones.

The library assumes that, for multiple processor machines, communication between processes running on the same node is more efficient. It binds coprocesses to cluster nodes in a way that minimizes internode communications.

The FD library supports Cartesian, cylindrical, and spherical coordinate systems as well as both periodic and nonperiodic boundaries.

Using C++ expression templates,[3] users can write C-like expressions such as $A = B \times C + D$ that manipulate grid objects. In contrast to the straightforward object-oriented approach that creates intermediate objects, expression templates offer much higher efficiency.

Finally, the FD library directly supports standard fixed and free boundary conditions.

Example

An example uses the CPE FD library to solve the same two-factor European exchange option as in the MC simulation example. The partial differential equation (PDE) formulation for any derivative c of two underlying stocks S_1 and S_2 is

$$-\frac{\partial c}{\partial t} = \frac{1}{2}\sum_{i=1}^{2}\sum_{i=1}^{2}\rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 c}{\partial S_i \partial S_j} + \sum_{i=1}^{2}\mu_1 S_i \frac{\partial c}{\partial S_i} - rc$$

The boundary condition for the PDE formulation is

$$c(S_1, 0, t) = S_1$$
$$c(0, S_2, t) = 0$$

The FD library discretizes the initial values on a grid and uses a difference equation to evolve the solution in time. It obtains two difference equations by transforming the original equation using the iterative Crank-Nicholson (ICN) scheme.[7] The first difference equation is shown in Figure 1a. The second difference equation is similar to this example.

To implement this scheme, the user codes the time-step propagation and boundary conditions. CPE automatically determines the parallelization scheme and handles ghostzone synchronization.

In Figure 1b, which shows the code expressing the first difference equation, grid is a Grid-class object whose constructor specifies a two-dimensional Cartesian coordinate system and a ghostzone of width 1 along each axis. This difference equation computes the grid values for the intermediate time step ($t + 1/2$, denoted by the index Th) from the previous time step (t, denoted by the index Tp).

The inner method specifies that the difference equation applies only to the nonboundary grid points. The shift method facilitates the access of neighboring grid points. For instance, shift(−1, 0) corresponds to $G_{x-1,y}$ in the difference equation. As this example demonstrates, there is a direct mapping between the difference equation and the code. Looping is not required because the FD library manipulates an entire grid object.

REMOTE EXECUTION

The CPE remote execution module ParaConnect decouples front-end applications from back-end parallel computation. Using ParaConnect, users can invoke parallel computations on a cluster from any machine on the same network or even over the Internet.

ParaConnect is functionally similar to the common object request broker architecture (Corba); the major difference is that it directly supports MPI-based parallel execution. Developers can use ParaConnect to design remote parallel objects that hold state and data, which persist in the back-end platform across multiple method invocations. Handles in the front-end client reflect entities in the back end and provide both blocking and non-blocking APIs to control the remote execution. Further, remote methods can push intermediate results to the front end at any time.

$$G_{x,y}^{t+\frac{1}{2}} = \frac{1}{2}\Bigg\{ G_{x,y}^{t} + \Delta t \Bigg[G_{x,y}^{t}\left(-\left(\frac{\sigma_x}{\Delta x}\right)^2 x^2 - \left(\frac{\sigma_y}{\Delta y}\right)^2 y^2 - r \right) +$$

$$\frac{1}{2}\Bigg(G_{x-1,y}^{t}\left(\left(\frac{\sigma_x}{\Delta x}\right)^2 x^2 - \frac{\mu_x}{\Delta x} x\right) + G_{x+1,y}^{t}\left(\left(\frac{\sigma_x}{\Delta x}\right)^2 x^2 + \frac{\mu_x}{\Delta x} x\right) +$$

$$G_{x,y-1}^{t}\left(\left(\frac{\sigma_y}{\Delta y}\right)^2 y^2 - \frac{\mu_y}{\Delta y} y\right) + G_{x,y+1}^{t}\left(\left(\frac{\sigma_y}{\Delta y}\right)^2 y^2 + \frac{\mu_y}{\Delta y} y\right)\Bigg) +$$

$$\frac{\rho_{xy}}{4}\frac{\sigma_x}{\Delta x}\frac{\sigma_y}{\Delta y} xy \left(G_{x-1,y-1}^{t} + G_{x+1,y+1}^{t} - G_{x-1,y+1}^{t} - G_{x+1,y-1}^{t} \right) \Bigg] +$$

$$G_{x,y}^{t} \Bigg\}$$

(a)

```
#define XC grid.coordinates(X)
#define YC grid.coordinates(Y)
#define sq(x) ((x) * (x))
...
grid[Th].inner() = 0.5 * (inner(
  grid[Tp] + dt * (
    grid[Tp] * (-(sq(sigmax)/sq(dx))*sq(XC) - (sq(sigmay)/sq(dy))*sq(YC) - r) +
    0.5 * (grid[Tp].shift(-1,0) * (sq(sigmax)/sq(dx)*sq(XC) - mux/dx*XC) +
           grid[Tp].shift(1,0) * (sq(sigmax)/sq(dx)*sq(XC) + mux/dx*XC) +
           grid[Tp].shift(0,-1) * (sq(sigmay)/sq(dy)*sq(YC) - muy/dy*YC) +
           grid[Tp].shift(0,1) * (sq(sigmay)/sq(dy)*sq(YC) + muy/dy*YC) ) +
    0.25*rhoxy*sigmax*sigmay*XC*YC/(deltax*deltay) * (
         grid[Tp].shift(-1,-1) + grid[Tp].shift(1,1) -
         grid[Tp].shift(-1,1) - grid[Tp].shift(1,-1) ) ) +
grid[Tp], grid.ghostzone() ) );
```

(b)

Figure 1. Finite-difference calculation. (a) One of two difference equations obtained via the iterative Crank-Nicholson scheme. (b) Code expressing the difference equation.

ParaConnect uses a driver-based approach for different remote access startup and MPI-process-launching mechanisms. Currently available start-up drivers support remote access using SSH, TCP, and TCP/SSL. MPI execution drivers support MPICH (www-unix.mcs.anl.gov/mpi/mpich/index.htm) and LAM/MPI (www.lam-mpi.org) with or without the Portable Batch System batch queue system.

The remote execution module uses the XML-RPC protocol internally for requests and responses. In addition, it uses base64 encoding for efficiently embedding data as it avoids the overhead associated with generating and parsing full XML representations.

CPE also provides an interface that allows remote parallel execution from a Microsoft Excel spreadsheet. Users can use menus to set up interfaces to back-end objects without any programming. They can also control configuration and startup of remote execution, parallel object creation and destruction, and method invocation and cancellation, all within Excel.

Moreover, changing the values of cells registered as remote method inputs reinvokes the remote methods, with the results going to spreadsheets when they are available. The underlying implementation uses ActiveX Control and Visual Basic for Application. Dynamic data exchange provides nonblocking method invocation and pushing of messages and output arguments.

EXPERIMENTAL RESULTS

We used the MC library in CPE to price several interest-rate derivative products under a

- three-factor cross-currency Hull-White (HW3) model,[6] and

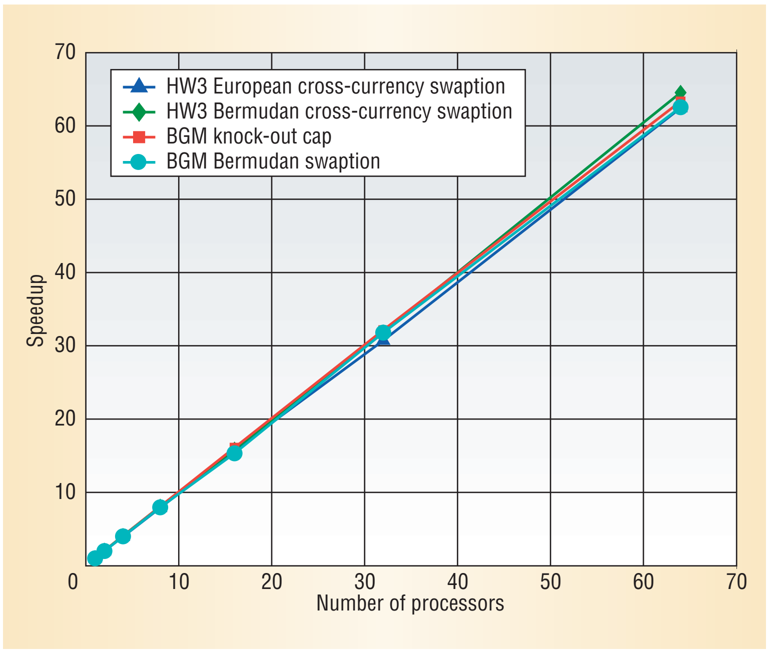

Figure 2. MC scaling graph for four applications (Gigabit Ethernet). The MC library achieves nearly linear scaling up to 64 processors.

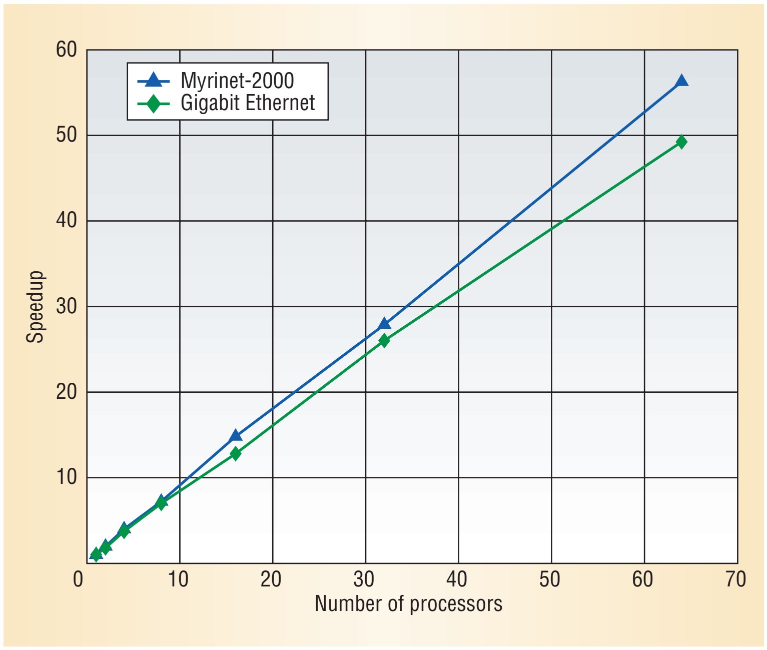

Figure 3. FD scaling graph for an HW3 application. The low-latency Myrinet interconnect reduces the time required for coprocesses to exchange ghostzone values, yielding better scaling.

- three-factor forward-rate model, also known as the Brace-Gatarek-Musiela (BGM) model.[8]

Our studies benchmarked four applications on a 32-node dual Intel Pentium Xeon Linux-based cluster with a Gigabit Ethernet interconnect. Figure 2 shows the relative performance for different numbers of processors, with nearly linear scaling up to 64 processors.

We also used the FD library to price an interest-rate product using the ICN scheme under an HW3 model and benchmarked this application using both Gigabit Ethernet and low-latency Myrinet-2000 interconnects to obtain the graph in Figure 3. Because coprocesses must exchange ghostzone values every time step, a communications bottleneck arises. As the figure shows, a low-latency interconnect, such as Myrinet, reduces the time required for this data exchange, yielding better scaling.

The Clustertech parallel environment offers unprecedented efficiency in developing financial engineering applications and porting them to a wide range of parallel cluster and symmetric multiprocessing machines. CPE's object-oriented approach facilitates information hiding as well as code reuse. Templates enable the libraries to be independent of data types, and expression templates minimize the creation of unnecessary objects.

The main difficulty in parallel programming is managing data transfers. The Tx class provides a convenient unifying concept for transporting complex data structures over arbitrary protocols. Using a metacompiler to automatically generate interfacing code produces simplified code and frees the programmer from dealing with low-level issues.

In the MC library, users need only provide code that describes the computation of a single sample; in the FD library, a distributed array abstraction can directly map difference equations to parallel grid operations. In both cases, the library automatically handles parallelization issues, and programmers only have to describe their computation's application-specific aspects.

The remote execution module provides an API for developing desktop-based applications that use a parallel machine. With the CPE remote execution library, users can access the parallel machine, create objects, invoke methods, and destroy objects via handles.

Together these abstractions make the task of developing parallel programs easier, leading to improvements in both productivity and code reliability. ■

Acknowledgment

We gratefully acknowledge support from the Hong Kong Innovation and Technology Commission (project no. UIM/68).

References

1. G. Allen et al., "Solving Einstein's Equations on Supercomputers," *Computer*, Dec. 1999, pp. 52-58.
2. A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.
3. T. Veldhuizen, "Expression Templates," *C++ Report*, vol. 7, no. 5, 1995, pp. 26-31.
4. S. Chiba, "A Metaobject Protocol for C++," *Proc. 10th Ann. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, 1995, pp. 285-299.
5. H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods,* Soc. for Industrial and Applied Mathematics, 1992, p. 241.
6. J.C. Hull, *Options, Futures and Other Derivatives*, 5th ed., Prentice Hall, 2002.
7. B. Gustafsson, H-O. Kreiss, and J. Oliger, *Time Dependent Problems and Difference Methods*, John Wiley & Sons, 1996.
8. R. Rebonato, *Interest-Rate Option Models: Understanding, Analysing and Using Models for Exotic Interest-Rate Options,* 2nd ed., John Wiley & Sons, 1998.

***Monk-Ping Leong** is a senior technology manager at Cluster Technology Ltd. His research interests include network security, parallel computing, and field-programmable systems. Leong received a PhD in computer science and engineering from the Chinese University of Hong Kong. Contact him at mpleong@clustertech.com.*

***Chi-Chiu Cheung** is a project manager at Cluster Technology Ltd. His research interests include parallel computing, distributed computing, and software engineering. Cheung received an MPhil in computer science and engineering from the Chinese University of Hong Kong. Contact him at cheungcc@clustertech.com.*

***Chin-Wang Cheung** is an analyst programmer at Cluster Technology Ltd. His research interests include parallel computing, neural networks, and financial engineering. Cheung received an MSc in e-commerce technologies from the Chinese University of Hong Kong. Contact him at cwcheung@clustertech.com.*

***Polly P.M. Wan** is an analyst programmer at Cluster Technology Ltd. Her research interests include parallel computing and time series analysis. Wan received an MPhil in computer science and engineering from the Chinese University of Hong Kong. Contact her at pmwan@clustertech.com.*

***Ivan K.H. Leung** is an analyst programmer at Cluster Technology Ltd. His research interests include reconfigurable computing, cryptography, and parallel computing. Leung received an MPhil in computer science and engineering from the Chinese University of Hong Kong. Contact him at khleung@clustertech.com.*

***Winnie M.M. Yeung** is an analyst programmer at Cluster Technology Ltd. Her research interests include spatial database and parallel programming. Yeung received an MMath in computer science from the University of Waterloo. Contact her at mmyeung@clustertech.com.*

***Wing-Seung Yuen** is an analyst programmer at Cluster Technology Ltd. Her research interests include computer-aided design of VLSI circuits, algorithms, and parallel computing. Yuen received an MPhil in computer science and engineering from the Chinese University of Hong Kong. Contact her at wsyuen@clustertech.com.*

***Kenneth S.K. Chow** is the chief operating officer of Cluster Technology Ltd. His research interests include modeling of interest-rate derivatives and quantitative strategies in fixed income, currencies, and commodities. Chow received a PhD in physics from Cornell University. Contact him at chowskei@clustertech.com.*

***Kwong-Sak Leung** is a chair professor at the Chinese University of Hong Kong. His research interests include knowledge engineering, bioinformatics, and fuzzy logic applications. Leung received a PhD in engineering from the University of London. He is a senior member of the IEEE Computer Society and a member of the ACM. Contact him at ksleung@cse.cuhk.edu.hk.*

***Philip H.W. Leong** is a professor at the Chinese University of Hong Kong. His research interests include reconfigurable computing, parallel computing, and signal processing. Leong received a PhD in engineering from the University of Sydney. He is a senior member of the IEEE Computer Society. Contact him at phwl@cse.cuhk.edu.hk.*