Productive Programming in Chapel: A Computation-Driven Introduction

Background

Michael Ferguson and Lydia Duncan
Cray Inc,
SC15 November 15th, 2015
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
DARPA HPCS: High Productivity Computing Systems

- **Goal:** improve productivity by a factor of 10x
- **Timeframe:** summer 2002 – fall 2012
- Cray developed a new system architecture, network, software, …
  - this became the very successful Cray XC30™ Supercomputer Series

...and a new programming language: Chapel
Chapel Motivation

**Q:** Why doesn’t parallel programming have an equivalent to Python / Matlab / Java / C++ / (your favorite programming language here) ?

- one that makes it easy to quickly get codes up and running
- one that is portable across system architectures and scales
- one that bridges the HPC, data analysis, and mainstream communities

**A:** We believe this is due not to any particular technical challenge, but rather a lack of sufficient...

...long-term efforts
...resources
...community will
...co-design between developers and users
...patience

*Chapel is our attempt to change this*
Chapel's Implementation

● Being developed as open source at GitHub
  ● Licensed as Apache v2.0 software

● Portable design and implementation, targeting:
  ● multicore desktops and laptops
  ● commodity clusters and the cloud
  ● HPC systems from Cray and other vendors
  ● in-progress: manycore processors, CPU+accelerator hybrids, …
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials

1 EF – ~2018: Cray ____; ~10,000,000 Processors
- TBD
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

1 TF – 1998: Cray T3E; 1,024 Processors
- Modeling of metallic magnet atoms
- Fortran + MPI (Message Passing Interface)

1 PF – 2008: Cray XT5; 150,000 Processors
- Superconductive materials
- C++/Fortran + MPI + vectorization

1 EF – ~20__: Cray ____; ~10,000,000 Processors
- TBD
- TBD: C/C++/Fortran + MPI + OpenMP/OpenACC/CUDA/OpenCL?

Or, perhaps something completely different?
STREAM Triad: a trivial parallel computation

**Given:** \( m \)-element vectors \( A, B, C \)

**Compute:** \( \forall i \in 1..m, A_i = B_i + \alpha \cdot C_i \)

**In pictures:**

\[
\begin{align*}
A & \quad \quad = \\
B & \quad + \\
C & \quad \alpha
\end{align*}
\]
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A, B, C$

**Compute:** $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel:
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A, B, C$

**Compute:** $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory):
STREAM Triad: a trivial parallel computation

**Given:** $m$-element vectors $A, B, C$

**Compute:** $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):
#include <hpcc.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }

    scalar = 3.0;

    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
#include <hpcc.h>
#endif
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d). \n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }

    scalar = 3.0;

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        a[j] = b[j]+scalar*c[j];
        HPCC_free(c);
        HPCC_free(b);
        HPCC_free(a);
    }

    return 0;
}
STREAM Triad: MPI+OpenMP vs. CUDA

**MPI + OpenMP**

```c
#include <hpcc.h>
#include <omp.h>

#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;
    cudaMalloc((void**)&d_a, sizeof(float)*N);
    cudaMalloc((void**)&d_b, sizeof(float)*N);
    cudaMalloc((void**)&d_c, sizeof(float)*N);
    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x);
    cudaThreadSynchronize();
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}

__global__
void set_array(float *a, float value, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) a[idx] = value;
}

__global__
void STREAM_Triad(float *a, float *b, float *c, float scalar, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) c[idx] = a[idx] + scalar * b[idx];
}
```

**CUDA**

```c
#include <hpcc.h>

#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;
    cudaMemcpy((void**)&d_a, sizeof(float)*N);
    cudaMemcpy((void**)&d_b, sizeof(float)*N);
    cudaMemcpy((void**)&d_c, sizeof(float)*N);
    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x);
    cudaThreadSynchronize();
    cudaMemcpy(d_a, (void**)&d_a, sizeof(float)*N);
    cudaMemcpy(d_b, (void**)&d_b, sizeof(float)*N);
    cudaMemcpy(d_c, (void**)&d_c, sizeof(float)*N);
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);
    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality.
Why so many programming models?

HPC has traditionally given users...
...low-level, control-centric programming models
...ones that are closely tied to the underlying hardware
...ones that support only a single type of parallelism

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>OpenMP / pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>Open[MP</td>
<td>CL</td>
</tr>
</tbody>
</table>

**benefits:** lots of control; decent generality; easy to implement
**downsides:** lots of user-managed detail; brittle to changes
Prototypical Next-Gen Processor Technologies

Intel MIC

Nvidia Echelon

AMD APU

Tilera Tile-Gx

Sources:
http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg
http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf
General Characteristics of These Architectures

- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past
 Rewinding a few slides...

**MPI + OpenMP**

```c
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    if (myRank == 0)
        rv = HPCC_Stream( params, 0 == myRank);
    MPI_BdtRedC( argv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

a = HPCC_XMALLOC( double, VectorSize );
b = HPCC_XMALLOC( double, VectorSize );
c = HPCC_XMALLOC( double, VectorSize );

scalar = 3.0;

#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
b[j] = 2.0;
c[j] = 1.0;
}

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
return 0;
```

**CUDA**

```c
#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;

    cudaMalloc((void**)&d_a, sizeof(float)*N);
cudaMalloc((void**)&d_b, sizeof(float)*N);
cudaMalloc((void**)&d_c, sizeof(float)*N);

    dim3 dimBlock(128);
    dim3 dimGrid( N / dimBlock.x );
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaThreadSynchronize();
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality.
**STREAM Triad: Chapel**

```chapel
config const m = 1000,
    alpha = 3.0;

const ProblemSpace = {1..m};

var A, B, C: [ProblemSpace] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;
```

**Philosophy:** Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Outline

✓ Motivation

➤ Chapel Background and Themes
- Learning the Base Language with n-body
- Short Introduction to Task Parallelism
- Hands-On 1: Hello World
- Short Introduction to Locality
- Data Parallelism with Jacobi
- Hands-On 2: Mandelbrot
- Project Status, Next Steps
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user’s program

- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

...target any parallelism available in the hardware

- **Types:** machines, nodes, cores, instruction

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>Chapel</td>
<td>task (or executable)</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>Chapel</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>Chapel</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>Chapel</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

Copyright 2015 Cray Inc.
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[(\frac{\text{vector}}{2}) + \frac{\text{vector}}{2}\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```chapel
proc main() {
    var n = 1000;
    var A, B: [1..n] real;
    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```chapel
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
    var A, B: [0..myN+1] real;
    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    }
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    }
    forall i in 1..myN do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;
  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```
proc main() {
  var n = 1000;
  var p = numProcs(),
        me = myProc(),
        myN = n/p,
        myLo = 1,
        myHi = myN;
  var A, B: [0..myN+1] real;
  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  } else
    myHi = myN-1;
  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  } else
    myLo = 2;
  forall i in myLo..myHi do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
A language may support both global- and local-view programming — in particular, Chapel does

```chapel
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

proc MySPMDProgram(myImageID, numImages) {
    ...
}
```
3) Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”
3) Multiresolution Design

**Multiresolution Design**: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control

*Chapel language concepts*

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Copyright 2015 Cray Inc.
4) Control over Locality/Affinity

Consider:
- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:
- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:
- Over time, we expect locality to matter more and more within the compute node as well
Shared Memory Programming Models

e.g., OpenMP, Pthreads

+ support dynamic, fine-grain parallelism
+ considered simpler, more like traditional programming
  ● “if you want to access something, simply name it”
  – no support for expressing locality/affinity; limits scalability
  – bugs can be subtle, difficult to track down (race conditions)
  – tend to require complex memory consistency models
Message Passing Programming Models

e.g., MPI

+ a more constrained model; can only access local data
+ runs on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
+ is relatively easy to implement
+ can serve as a strong foundation for higher-level models
+ users have been able to get real work done with it
Message Passing Programming Models

e.g., MPI
- communication must be used to get copies of remote data
  - tends to reveal too much about how to transfer data, not simply what
- only supports “cooperating executable”-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches between sends/recvs, buffer overflows, etc.
Hybrid Programming Models

e.g., MPI+OpenMP/Pthreads/CUDA, UPC+OpenMP, …

+ supports a division of labor: each handles what it does best
+ permits overheads to be amortized across processor cores, as compared to using MPI alone

– requires multiple notations to express a single logical parallel algorithm, each with its own distinct semantics
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:
- support a shared namespace on distributed memory
- permit any parallel task to access any lexically visible variable
- doesn’t matter if it’s local or remote
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn’t matter if it’s local or remote
- establish a strong sense of ownership
  - every variable has a well-defined location
  - local variables are cheaper to access than remote ones
Chapel and PGAS

- Chapel is a PGAS language…

…but unlike most, it’s not restricted to SPMD

⇒ never think in terms of “the other copies of the program”

Locales (think: “compute nodes”)

0 1 2 3 4
Traditional PGAS Languages

PGAS founding members: Co-Array Fortran, UPC, Titanium

- extensions to Fortran, C, and Java, respectively
- details vary, but potential for:
  - arrays that are decomposed across compute nodes
  - pointers that refer to remote objects
- note that earlier languages could arguably also be considered PGAS, but the term hadn’t been coined yet
### PGAS: What’s in a Name?

<table>
<thead>
<tr>
<th>Trad. PGAS Languages</th>
<th>memory model</th>
<th>programming model</th>
<th>execution model</th>
<th>data structures</th>
<th>communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>distributed memory</td>
<td>cooperating executables (often SPMD in practice)</td>
<td>manually fragmented</td>
<td>APIs</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>shared memory</td>
<td>global-view parallelism</td>
<td>shared memory multithreaded</td>
<td>shared memory arrays</td>
<td>N/A</td>
</tr>
<tr>
<td>Chapel</td>
<td>PGAS</td>
<td>global-view parallelism</td>
<td>distributed memory multithreaded</td>
<td>global-view distributed arrays</td>
<td>implicit</td>
</tr>
<tr>
<td>Titanium</td>
<td>PGAS</td>
<td>Single Program, Multiple Data (SPMD)</td>
<td></td>
<td>co-arrays</td>
<td>co-array refs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1D block-cyc arrays/ distributed pointers</td>
<td>implicit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>class-based arrays/ distributed pointers</td>
<td>method-based</td>
</tr>
</tbody>
</table>

**Trad. PGAS Languages:**
- CAF
- UPC
- Titanium
- Chapel

**APIs:**
- Co-arrays
- Co-array refs
- 1D block-cyc arrays
- Distributed pointers
- Method-based

**Communication Models:**
- APIs
- N/A
Traditional PGAS Languages

e.g., Co-Array Fortran, UPC

+ support a shared namespace, like shared-memory
+ support a strong sense of ownership and locality
  • each variable is stored in a particular memory segment
  • tasks can access any visible variable, local or remote
  • local variables are cheaper to access than remote ones
+ implicit communication eases user burden; permits compiler to use best mechanisms available
Traditional PGAS Languages

e.g., Co-Array Fortran, UPC
- restricted to SPMD programming and execution models
- data structures not as flexible/rich as one might like
- retain many of the downsides of shared-memory
  - error cases, memory consistency models
Next-Generation PGAS Languages

e.g., Chapel (also Charm++, X10, Fortress, …)
  - breaks out of SPMD mold via global multithreading
  - richer set of distributed data structures
    - retains many of the downsides of shared-memory
      - error cases, memory consistency models
5) Reduce HPC ↔ Mainstream Language Gap

Consider:

- Students graduate with training in Java, Matlab, Python, etc.
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:

- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.