Cascade

Burton Smith
Cray Inc.

The DARPA HPCS Program

- DARPA’s High Productivity Computer Systems program has three central objectives:
  - Improve HPC system productivity
  - Improve HPC programmer productivity
  - Improve system robustness (reliability and security)
- Three phases are planned:
  - Phase 1 (7/02–7/03): Define a system concept
    - Cray, HP, IBM, SGI, Sun
  - Phase 2 (7/03–7/06): Prepare a development plan
    - Cray, IBM, Sun
  - Phase 3 (7/06–7/10): Develop a system
    - Two awardees

Our approach to HPCS

- High system productivity
  - Implement *very* high global bandwidth
  - Use that bandwidth (the “wires”) well
  - Provide configurability to match user needs
- High human productivity and portability
  - Support a mixed UMA/NUMA programming model
  - Deliver strong compiler and runtime support
  - Pursue higher level programming languages
- System robustness
  - Virtualize all resources
  - Make all resources dynamically reconfigurable
  - Use introspection to detect bugs or intrusion

Bandwidth is expensive

- High global system bandwidth is a “good thing”
  - It determines performance for many problems
  - It also makes improved programmability possible
- Sadly, connection costs badly trail Moore’s law
  - Packages, circuit boards, wires, optical fibers…
  - Most of hardware cost is connection cost
- Cray builds systems with high global bandwidth
  - This strongly influences most of what we do
- Cray needs to stay competitive
  - We must make bandwidth cost less
  - We must use bandwidth wisely
- These ideas motivate much of Cascade’s architecture

Making bandwidth cost less

- Tune bandwidth (∴ cost) to match customer needs
- Use the cheapest link technology that fits each bill
  - Optical or electrical
- Make data rates as fast as the technology permits
  - Spending transistors in this cause is a bargain
- Design routers that use all the network links well
  - Randomized non-minimal routing, for example
- Use efficient network topologies
  - High degree routers

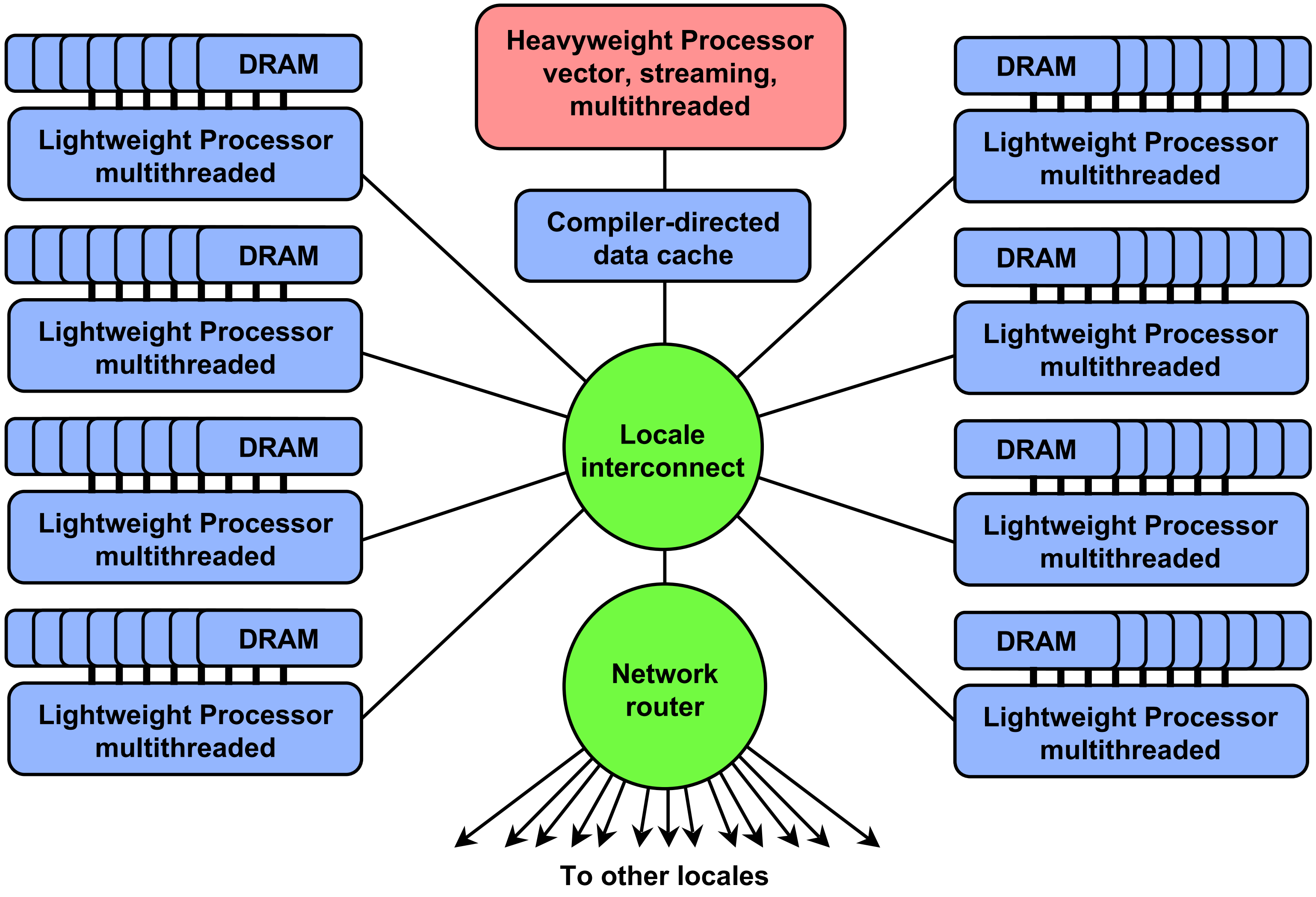
Network Topology

- If network link load is well balanced, node injection bandwidth B times average distance d (in hops) is bounded by link bandwidth β times node degree Δ
- Cost/node is proportional to β times Δ
  - Signaling rate and package pin count determine it
- Increasing the degree Δ lowers the average distance d
  - β can be lowered to maintain (or even improve) cost
  - B will increase as d decreases
- Conclusion: trading high node degree for link bandwidth can yield better injection bandwidth, latency, and cost

Using bandwidth wisely

- Implement shared memory (UMA/NUMA hybrid)
  - Eliminate overhead to enable small messages
- Tolerate memory latency
  - CC-NUMA wastes bandwidth moving data around
  - Use vectors and multithreading instead
- Exploit temporal locality in “heavyweight” processors
  - Compiler-directed data cache
  - Architectural support for streaming
- Exploit spatial locality in “lightweight” processors
  - Threads migrate to follow the spatial locality
- Use other types of locality whenever possible
  - *e.g.* atomic memory operations

A locale building block


DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded
Heavyweight Processor vector, streaming, multithreaded
Compiler-directed data cache
Locale interconnect
Network router
To other locales
DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded
DRAM
Lightweight Processor multithreaded

Compiler-directed data cache

- The compiler often knows if data are safely cacheable, *i.e.* are temporarily private or temporarily constant
- It can tell the hardware what data to cache and when to flush or simply invalidate it
  - Dead values as well as constants are invalidated
- Unnecessary coherence traffic is eliminated
- Latency and network bandwidth demand are reduced
- Threads within a processor can communicate and synchronize within cache to exploit streaming locality
- The cache becomes a much more general tool for exploiting many forms of temporal locality

Lightweight threads in memory

- Lightweight threads in the memory can exploit spatial locality by migrating to memory they refer to
  - Some remote references just block the thread
  - Others cause migration to the remote memory
- Memory is block-hashed to provide a compromise between spatial locality and reference distribution
- Processor-in-memory (PIM) technology is an ideal implementation vehicle for this idea
- Threads are spawned by sending parcels to memory from either heavyweight or other lightweight threads
  - Spawning and migration overheads are minimal
  - In-memory operations are handled specially
- Generally, the compiler packages temporally local loops for heavy threads and the rest for light ones

Sparse matrix-vector product

```
int nrows, rowp[];
val_idx_pair a[];
double b[], c[];
for (int i = 0; i<nrows; i++) {
  double sum = 0.0;
  for(int k = rowp[i]; k < rowp[i+1]; k++){
    sum += a[k].val*b[a[k].idx];
  }
  c[i]=sum;
}
```

The data layout:

$$\begin{array}{lccccccc} & & & \texttt{rowp[i]} \searrow & & & \texttt{rowp[i+1]} \searrow & \\ \texttt{a[k].val:} & . & . & . & a_{i,15} & a_{i,42} & a_{i,53} & . \; . \; . \\ \texttt{a[k].idx:} & . & . & . & 15 & 42 & 53 & . \; . \; . \end{array}$$

Lightweight threads on SparseMV

- There are three memory references for every two flops
  - Two memory references are local and one isn't
  - There are 2 flops per "global" memory reference
  - (Dense) inner product is the same if one or both vectors is unit stride
- The thread context required for this loop comprises:
  - The program counter (the "method")
  - Exception flags, *etc*. (perhaps packed with the PC)
  - The pointer to the vector of value-index pairs **`a[]`**
  - A limit for loop control, set to **`&rowp[i+1]`**
  - Two pointers to the vectors of doubles **`b[]`** and **`c[]`**
  - The double accumulator **`sum`**
  - A few temporaries, *e.g.* for **`b[a[k].idx]`**

Productive programming

- Matlab lets scientists be productive programmers
  - Execution performance is marginal at best
  - Manual translation to Fortran is the typical fix
- We are developing *Chapel*, a programming language aimed at both programmability and performance
- Its key features:
  - Interprocedural polymorphic type inference
  - Locality abstraction via first-class domains
  - Explicit parallel operations over domains
  - Implicit parallelism packaging and optimization
  - Automatic thread and memory management
  - Open-source implementation
- We will also support mixed-legacy-language programs
  - Fortran, C++, MPI, shmem, coarray languages

NAS CG conj_grad() in Chapel

```
function conj_grad(A, X): {
  const cgitmax = 25;

  var Z = 0.0;
  var R = X;
  var P = R;
  var rho = sum R**2;

  for cgit in (1..cgitmax) {
    var Q = sum(dim=2) (A*P);

    var alpha = rho / sum (P*Q);
    Z += alpha*P;
    R -= alpha*Q;

    var rho0 = rho;
    rho = sum R**2;
    var beta = rho / rho0;
    P = R + beta*P;
  }
  R = sum(dim=2) (A*Z);
  var rnorm = sqrt(sum (X-R)**2);

  return (Z, rnorm);
}
```

Parameter types elided (inferred from callsite)

Built-in array reductions

Sequential iteration over an anonymous domain

Partial array reductions

Global view ⇒ processors not exposed in computation, array sizes

Composable parallelism ⇒ this (parallel) function could be called from a parallel task (which in turn could be called from another…)

Function return type elided (inferred from return statement)

Local variable types elided (inferred from initializer, uses)

Whole-array operations ⇒ data parallel implementation

Operate on sparse arrays as though dense, and independently of implementing data structures

Separation of concerns ⇒ locale views, domain/array distributions & alignments, and sparse data structures are expressed elsewhere

Promotion of scalar operators, values, and functions

Support for tuples

Fortran+MPI	=	173-288 lines	(1265 tokens)
Chapel	=	20 lines	(150 tokens)

A few more Cascade tasks

- Operating system
  - Scalability, robustness, utility
- System infrastructure
  - RAS system, power, cooling
- Interconnect implementation
  - Router design, network topology
- Productivity assessment
  - Metrics, modeling, prediction, applications
- Implementation technology
  - Interconnect, chip packaging, power, cooling
- Debugging
  - Correctness, performance
- Marketing
  - Costs to develop and manufacture, sales outlook

Cascade collaborators

- Cray Inc.
  - Burton Smith, David Callahan, Steve Scott, . . .
- Caltech/JPL
  - Thomas Sterling, Hans Zima, Larry Bergman, . . .
- Notre Dame
  - Peter Kogge, Jay Brockman, . . .
- Stanford
  - Bill Dally, Christos Kozyrakis, . . .

Our experience base

The Cray team has experience in these technologies:

- Latency-tolerant vector NUMA systems
- Latency-tolerant multithreaded UMA systems
- Processor-in-memory technology
- High bandwidth interconnection networks
- High-productivity compiler technology
- Whole-program, incremental compilation
- Run-time systems for fine-grain synchronization
- Scalable, highly productive operating systems
- Supercomputer system integration

Conclusions

- HPCS matches Cray business objectives well
- We and our collaborators have expertise in the technological directions we intend to pursue
- We are confident of a successful outcome