PICS - a Performance-analysis-based Introspective Control System to Steer Parallel Applications

Yanhua Sun, Jonathan Lifflander, Laxmikant V. Kalé

April 29, 2014

Motivation

Complexity

Modern parallel computer systems are becoming extremely complex due to complicated network topologies, hierarchical storage systems, heterogeneous processing units, etc.
Obtaining best performance is challenging

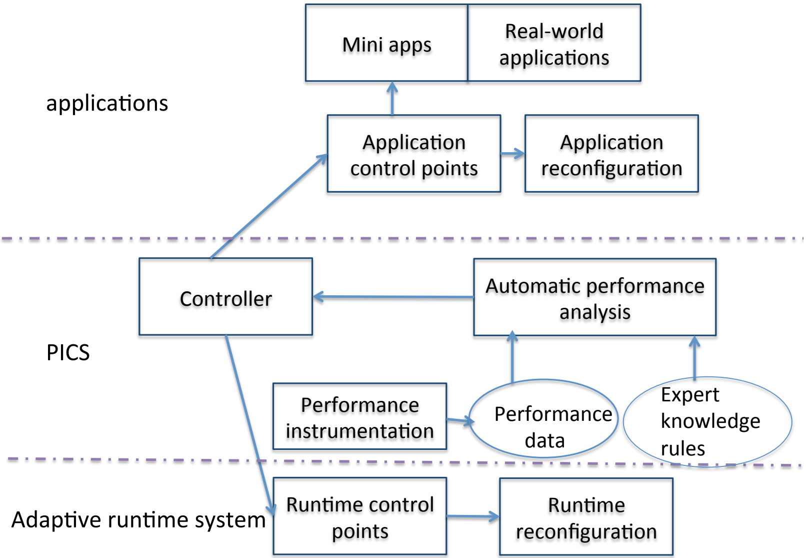
Applications and runtime should be reconfigurable to adapt to various situations

The goal of the control system is to adjust the configuration automatically based on application-specific knowledge and runtime observations.

Outline

- Overview of PICS framework
- Control points in the runtime system and applications
- Automatic performance analysis to speedup tuning
- APIs implemented in Charm++
- Results of benchmarks and applications

Overview of PICS framework



Control Points

Control points

Control points are tunable parameters for application and runtime to interact with control system. First proposed in Dooley's research.

1. Name, Values : default, min, max
2. Movement unit: $+1, \times 2$
3. Effects, directions
   - Degree of parallelism
   - Grainsize
   - Priority
   - Memory usage
   - GPU load
   - Message size
   - Number of messages
   - other effects

Application Control Points

1. Application specific control points provided by users
2. Applications should be able to reconfigure to use new values

Control points	Effects	Use Cases
sub-block size	parallelism, grain size	Jacobi, Wave, stencil code
parallel threshold	parallelism, overhead, grain size	state space search
stages in pipeline	number of messages, message size	pipeline collectives
algorithm selection	degree of parallelism, grain size	3D FFT decomposition (slab or pencil)
software cache size	memory usage, amount of communication	ChaNGa
ratio of GPU CPU load	computation, load balance	NAMD, ChaNGa

Runtime System Control Points

1. Traditionally, configurations for the runtime system do not change
2. Configurations for the runtime system itself should be tunable

1. Registered by runtime itself
2. Requires no change from applications
3. Affect all applications

Runtime System Control Points

Control points	Effects	Use Cases
broadcast algorithm selection	communication	most applications
broadcast/reduction branch factor	critical path	most applications(NAMD)
compression algorithm	communication, overhead	NAMD, ChaNGa
fault tolerance frequency	overhead, memory usage	most applications
load balancing frequency	overhead, load balance	most applications
tracing data disk write frequency	memory usage, overhead	most applications
number of AMPI virtual threads	grain size	AMPI applications

Observe Program Behaviors

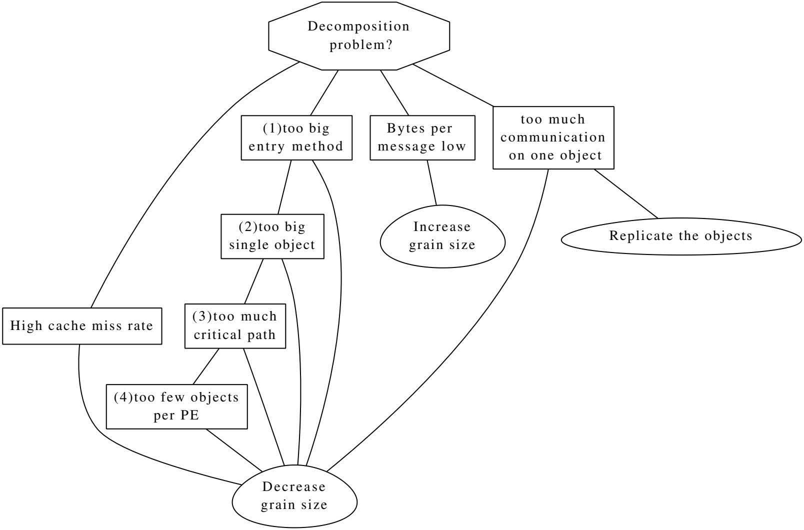
- Record all events
  - Events : begin idle, end idle
  - Functions: name, begin execution, end execution
  - Communication : message creation, size, source/destination
  - Hardware counters
- Module link, no source code modification
- Performance summary data

Automatically Analyze the Performance

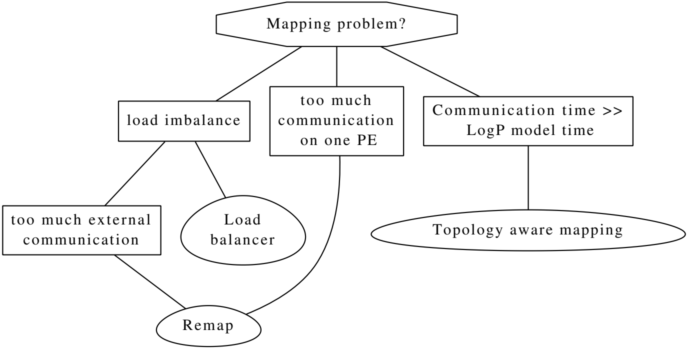
Many control points are registered. How to reduce the search space?
Performance Analysis - Identify Program Problems

- Decomposition
- Mapping
- Scheduling

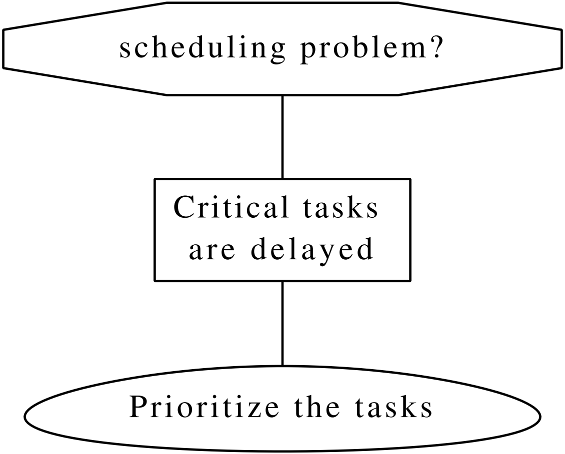
Decomposition Characteristics



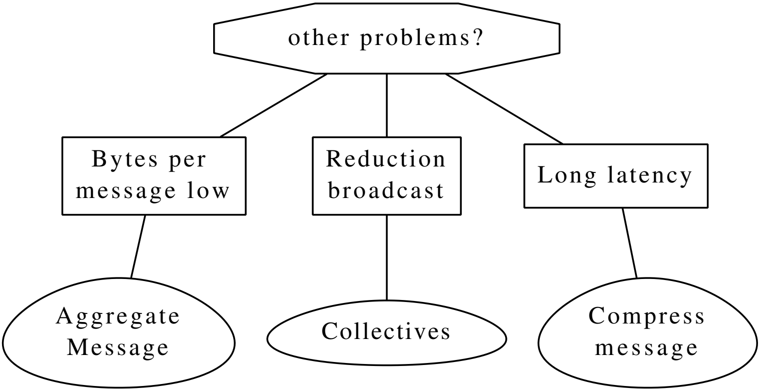
Mapping Characteristics



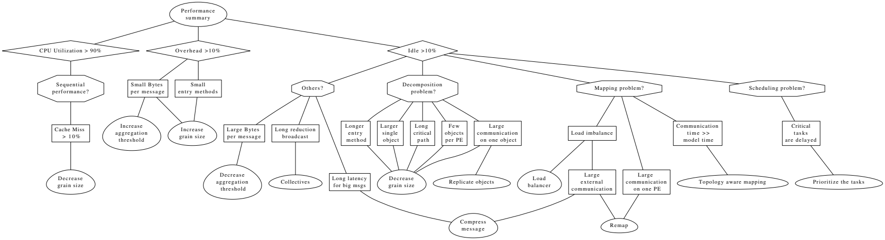
Scheduling Characteristics



Other Characteristics



Correlate Performance with Control Points



- One box can have multiple children
- One egg can have multiple parents

Correlate Performance with Control Points

Traverse the tree using the performance summary results

- performance results ⇒ solutions
- solution ⇒ effect of control points
- What control points to tune, in which direction!
- How much?
  - grainsize : $\frac{MaxObjLoad}{AvgLoad}$
- Feed results into the control points database

Control System APIs

```
typedef struct ControlPoint_t
{
    char      name[30];
    enum      TP_DATATYPE datatype;
    double    defaultValue;
    double    currentValue;
    double    minValue;
    double    maxValue;
    double    bestValue;
    double    moveUnit;
    int       moveOP;
    int       effect;
    int       effectDirection;
    int       strategy;
    int       entryEP;
    int       objectID;
}ControlPoint;
```

APIs for applications

```
void registerControlPoint(ControlPoint *tp);

void startStep();
void endStep();

void startPhase(int phaseId);
void endPhase();

double getTunedParameter(const char *name, bool *valid);
```

Experimental Results of Benchmarks and Applications

1. Control points
2. Performance problem
3. Bluegene/Q machine, Cray XE6 machine

Tuning Message Pipeline

- Control point: number of pipeline messages

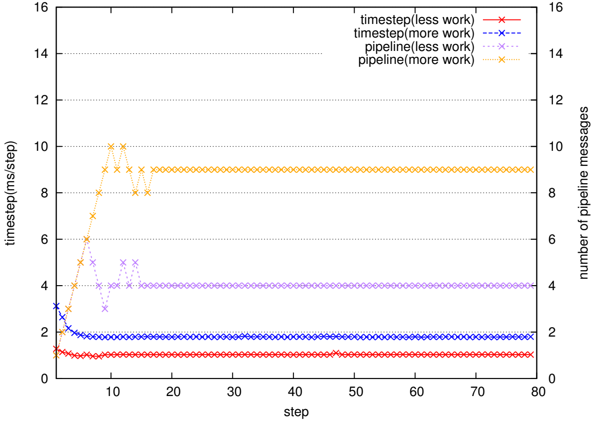


Figure: Tuning the number of pipeline messages

Message Compression

- Control points: compression algorithm for each type message
- Runtime control points

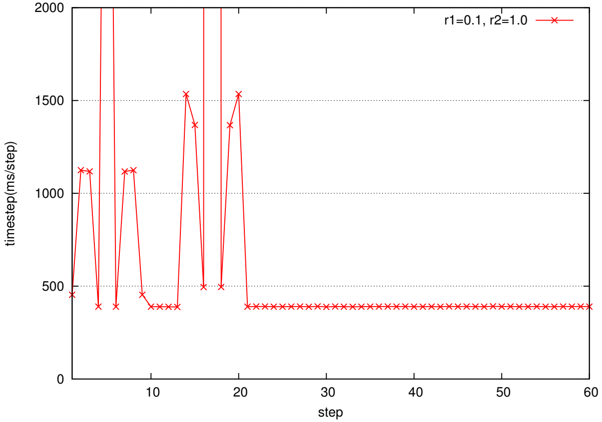


Figure: Steering the compression algorithm for all-to-all benchmark

Jacobi3d Performance Steering

- Control Points: sub-block size in each dimension
- Three control points
- Cache miss rate, high idle suggest decreasing sub-block size
- Overhead

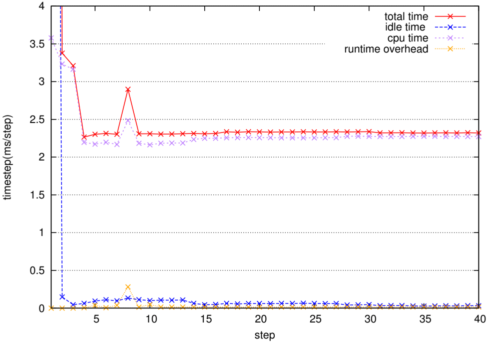


Figure: Jacobi3d performance steering on 64 cores for problem of 1024*1024*1024

Communication Bottleneck in ChaNGa

- Control points: number of mirrors
- Ratio of maximum communication per object to average

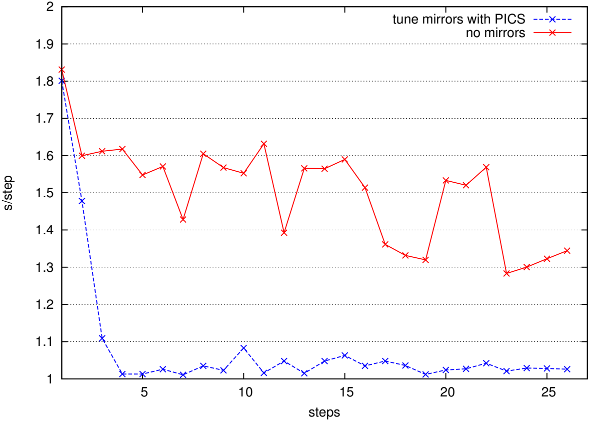


Figure: Time cost of calculating gravity for various mirrors and no mirror on 16k cores on Blue Gene/Q

Conclusion

- Automatic performance tuning is required to improve productivity and performance
- Automatic performance analysis helps guide performance steering
- Steering both runtime system and applications are important