Towards automatic performance analysis of parallel programs

Amitabh B. Sinha

Outline of talk

- Introduction
  - automatic performance analysis
- Automatic performance analysis of Charm programs
  - knowledge of program through language constructs and libraries
  - performance analysis techniques
  - integrated tool for automatic analysis
  - case study in parallel molecular dynamics
- Automatic performance analysis of parallel queries
  - basic operations and parallelization

Introduction

- Performance feedback is necessary

- Current work in performance feedback
  - visual feedback about processor utilization, etc.
  - often require manual instrumentation
  - user has lots of data to examine to detect problems

- Need automatic performance analysis
  - e.g., given a program in which processes have imbalanced load, the performance analysis system should detect and report this to the user

Introduction: automatic analysis

- Automatic analysis is feasible
  - Small set of commonly occurring problems

- How does one typically do performance analysis?
  - analysis ← techniques ← program behavior
  - e.g., load imbalance ← balance analysis ← processor loads

- How is <u>automatic</u> analysis feasible?
  - acquire information about program behavior
  - acquisition must be <u>automatic</u>
  - use information to apply standard techniques
  - application must be <u>automatic</u>

Introduction: automatic analysis

- What program behavioral characteristics are needed?
  - sub-tasks (placement and granularity)
  - communication (messages, locks, and disk i/o)
- How is information about program behavior acquired?
  - knowledge of the specific application
  - knowledge provided by the language through
    * compiler support (language constructs, annotations, and static analysis)
    * system libraries (barrier)

Charm

- Charm is portable across a wide variety of MIMD machines including IBM SP-2, NCUBE-II, CM-5, Paragon, Sequent, and clusters of workstations.
- Knowledge of program acquired through
  - language features
    - chares and branch office chares
    - information sharing abstractions
  - libraries
    - dynamic load balancing
    - queuing strategies
    - quiescence detection

Charm: language constructs

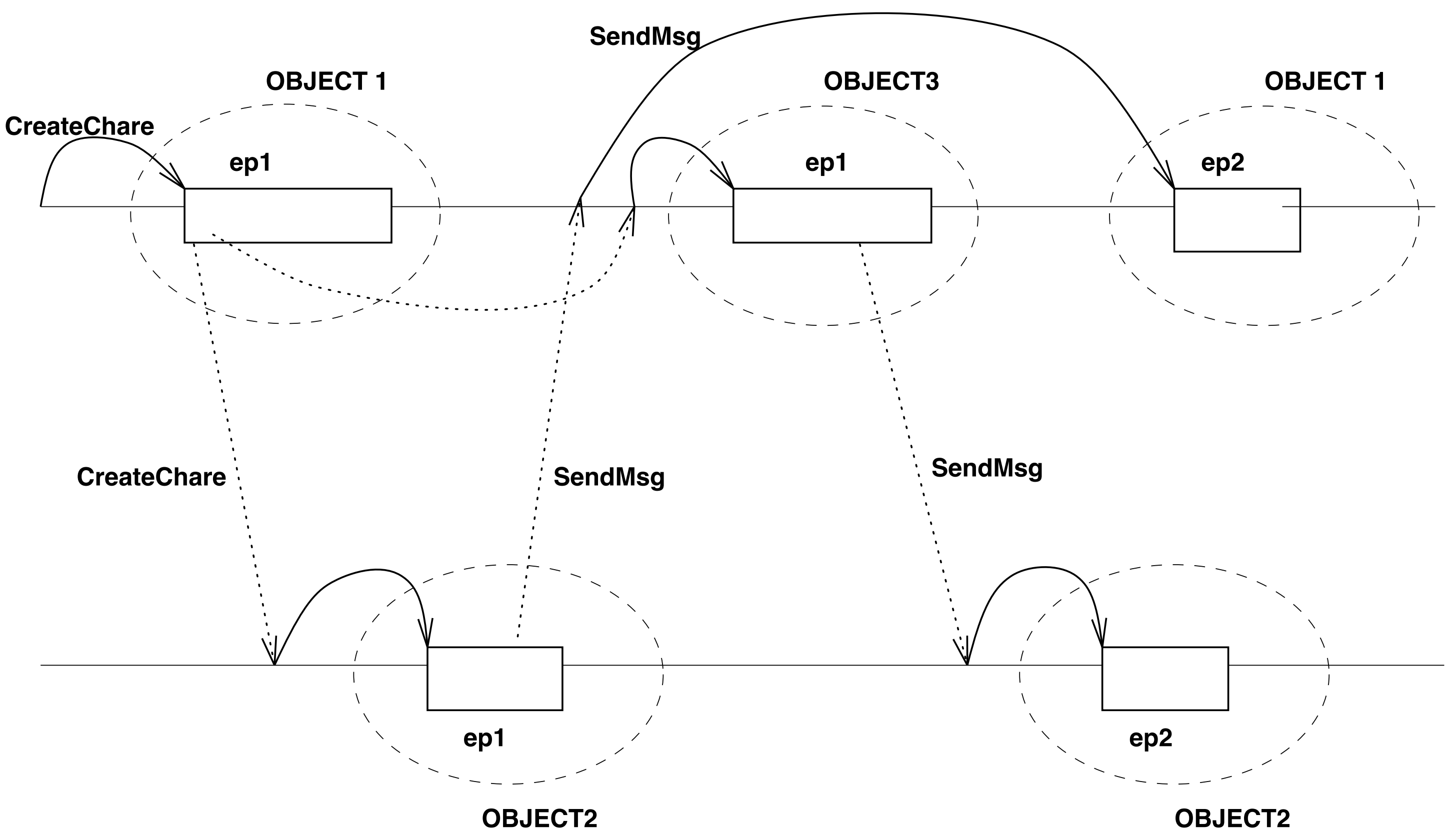
- Charm is object-based:

```
chare <CHARE1> {
    <data-area of chare>
    entry <EP1>: (message <type1> *m)
        C-code block
    ...
    private | public <name>()
        C-code block
}
```

- Message-driven execution model:
  - message contains address of entry method
  - execution of message automatically scheduled by system
  - the execution of each entry method is atomic

Charm: language constructs

- Sub-tasks (placement and granularity)



Message being sent, either to another processor or to self to be enqueued in the creation/response queue

A buffered message being picked up from the creation/response queue by the run-time system for execution.

Charm: language constructs

- Charm provides multiple modes of information sharing
  - each mode is an **adt** with known operators
  - interface to user is uniform across all machines
  - implementation can be and is machine specific
- Following modes are currently supported:
  - **Read Only / Write Once**
    - initialized once, and only read thereafter
  - **Accumulator**
    - operator is commutative associative, e.g., counter
  - **Monotonic**
    - updates are idempotent and monotonic, e.g., cost of best solution in branch&bound
  - **Distributed table**
    - each entry in table is a (key, data) pair
    - operators are Find, Insert, and Delete

Charm: system libraries

- Dynamic load balancing
  - user can choose a load balancing strategy at compile-time, e.g., random, ACWN, hierarchical, etc.
  - when a chare is created, it is placed under the control of the load balancing strategy
  - chares are moved freely around to balance load, and are created on least loaded processor
  - once a chare is created it is anchored to that processor; there is no migration

  - knowledge made available about the program
    * placement of tasks
    * computational demands of tasks

Charm: system libraries

- Queueing strategies
  - decides the order in which arriving messages are scheduled
  - prioritized queueing strategies
  - knowledge made available about the program
    * order of scheduling of messages
- Quiescence detection
  - detects a system state when there are
    * no more messages being processed, and
    * no messages waiting in queues
  - provides a mechanism for global synchronization
  - knowledge made available about the program
    * synchronization in the program

Projections: automatic analysis

- Automatic analysis is an iterative process
  - link program using "-execmode projections" option
  - execute program to produce traces automatically
  - use Projections to analyze traces
  - get analysis and change program, repeat
- Event graph
  - $V = \{v \mid v \text{ is a user event }\}$
  - For any $v \in V$,
    - v_c: time of creation
    - v_s: time system began executing it
    - v_f: time system finished executing it
  - $E = \{(x, y) \mid x, y \in V \text{ and } x \text{ created } y \ (x \rightarrow y) \ \}$
  - (V, E) defines the event graph

Automatic performance analysis: algorithm

```
Expert(V, E) {
    DetermineLogicalSeparationPoints(V, E);

    for each logical phase {
        utilization = ComputeEventCounts();
        if (utilization < 0.75) {
            SystemIdiosyncrasy();
            PhaseByPhaseAnalysis();
        }
    }
    EvaluateLDB();
    SharedVariableAnalysis();
}
```

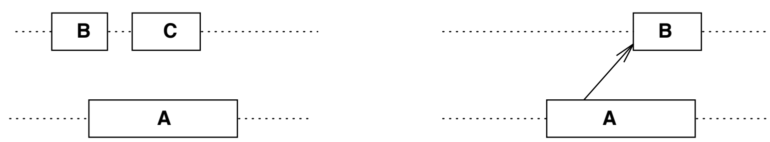
Automatic analysis: logical separation points

- What is the time interval for the analysis?
  - entire period of execution
  - equal intervals of time
  - user-specified
  - automatic

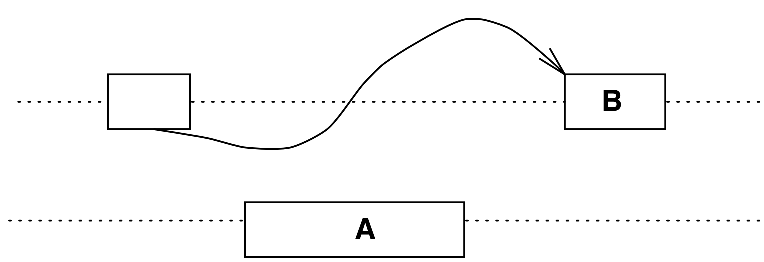
- How do you automatically decide meaningful intervals?
  - events that separate naturally repeating intervals
  - set of events whose performance does not affect performance of events after it

Automatic analysis: logical separation points

- What are logical separation points?
  - nothing else happens concurrently
    $(\neg\exists t)(((t_s \leq x_f) \wedge (t_f \geq x_s)) \wedge \neg(x \rightarrow t))$



  - no cross-over events (created before and processed after it)
    $(\neg\exists t)((t_c \leq x_f) \wedge (t_s \geq x_f) \wedge \neg(x \rightarrow t))$



- What are logically independent phases?

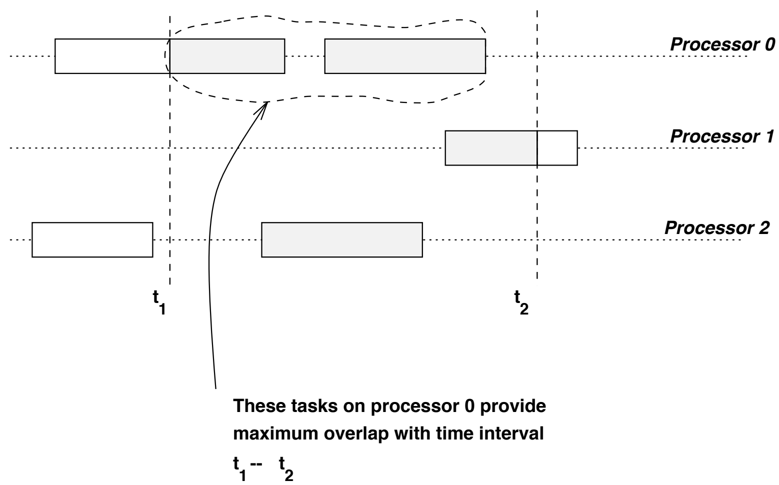
Automatic analysis: severity

- Motivation for severity analysis
  - all problems not equally severe
  - report problems in order of their effect on performance

Severity: The *severity* of a performance problem is the amount of reduction in the program's execution time if the problem is fixed.

Automatic analysis: severity

- Let the solution of a problem eliminate (t_1, t_2)
- Is severity $= t_2 - t_1$?
- Actually, severity $= t_2 - t_1 - overlap(t_1, t_2)$?



$$overlap(t_1, t_2) =$$

$$max\{\sum_{v \in V_p^{t_1,t_2}} (min(t_2, v_f) - max(t_1, v_s)) \mid p \in P\},$$

$$\text{where } V_p^{t_1,t_2} = \{v \mid (v \in V_p) \wedge (v_s \leq t_2) \wedge (v_f \geq t_1)\}.$$

ComputeEventCounts

N_e^p	number of instances of execution of the entry method e on processor p
N_e	number of instances of execution of the entry method e on all processors (i.e., $\sum_p N_e^p$)
G_e^p	average granularity for the entry method e on processor p
G_e	average granularity for the entry method e on all processors
T_e	total time spent executing entry method e across all processors (i.e., $N_e G_e$)

Utility analysis

- Is it useful to create a task (cost/utility)?
  - What is the cost of creating a task?

    cost of creating a task =

    the cost of creating message +
    cost of sending message across +
    cost of scheduling message

  - What is the utility of task?

    utility of task = granularity of entry method

- Severity of granularity problem for entry method x
  - acceptable granularity is A_x
  - new number of events of entry method x are $\frac{T_x}{A_x}$
  - new overhead $O_x \frac{T_x}{A_x}$
  - severity = $\frac{O_x N_x - O_x \frac{T_x}{A_x}}{P}$

Balance analysis

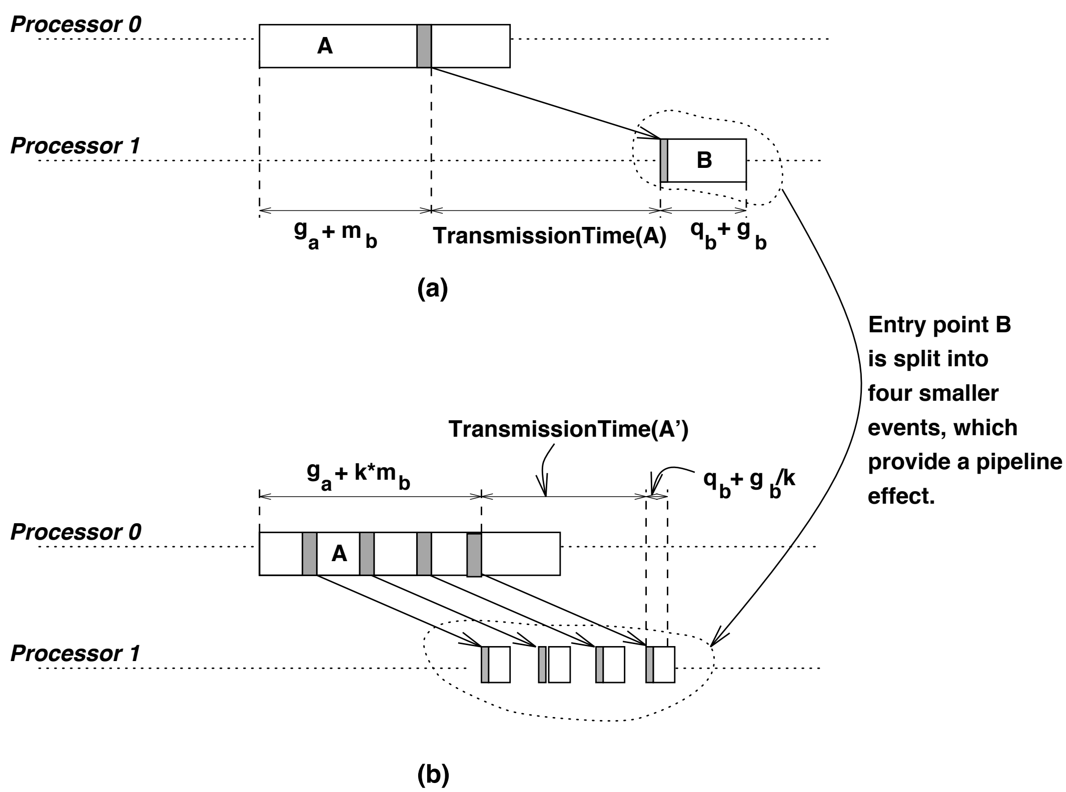
- Are user work, overheads, etc., balanced?
  - user work
  - overheads
  - user+overheads

- Severity of imbalance in number of events of entry method x
  - each processor gets equal work, i.e., $\frac{N_x}{P}$
  - processor having maximum work does $max(N_x^p) - \frac{N_x}{P}$ less work
  - severity = $(max(N_x^p) - \frac{N_x}{P})G_x$

- Severity of imbalance in granularity of entry method x
  - processor having maximum granularity does $max(G_x^p) - G_x$ less work
  - severity = $(max(G_x^p) - G_x)N_x^p$

Pipelining analysis

- When should you split a message into smaller ones?
  - when it arrives at an idle processor
  - large code block executes after it arrives



- severity $= (((g_a + m_b) + (\alpha + \beta s_b) + (g_b + q_b))$
  $-((g_a + km_b) + (\alpha + \frac{\beta s_b}{k}) + (\frac{g_b}{k} + q_b)))$
  $= (g_b + \beta s_b)(1 - \frac{1}{k}) - (k-1)m_b)$
  - solve differential equation for best $k = \sqrt{(\frac{\beta s_b + g_b}{m_b})}$
  - need to account for overlap

Shared variable analysis

- make a read-only/write-once variable which is accessed infrequently into an entry in a distributed table.
- make an entry in a distributed table, which is accessed very frequently by many different processors, into a write-once variable
- co-locate insertion and access for entries of a distributed table if they are accessed only once
- cache repeatedly accessed entries of a distributed table

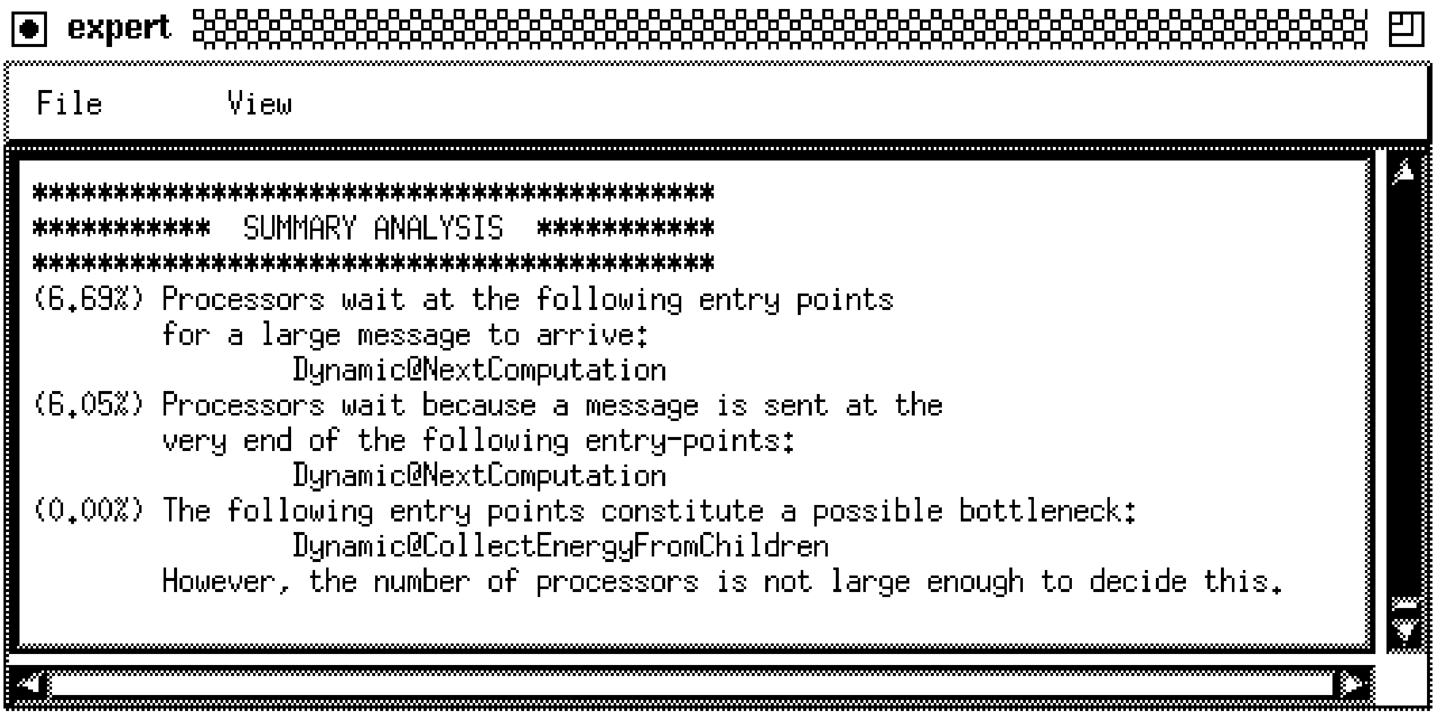
Case study: EGO

- Parallel molecular dynamics program
  - Coulomb forces between every pair of atoms
  - Bonded forces between atoms participating in a bond
  - Computationally intensive: $O(n^2)$ interactions for Coulomb forces
- How can computation of $O(n^2)$ interactions be reduced?
  - Newton's third law
  - distance classes

Case study: EGO

- Program flow
  - Distribute atoms equally across all processors
  - First, each processor computes interactions for atoms on itself
  - Next, each processor sends out a message:
    * coordinates of atoms it owns
    * forces on atoms it owns
  - Each processor computes interactions for atoms on itself with atoms in message

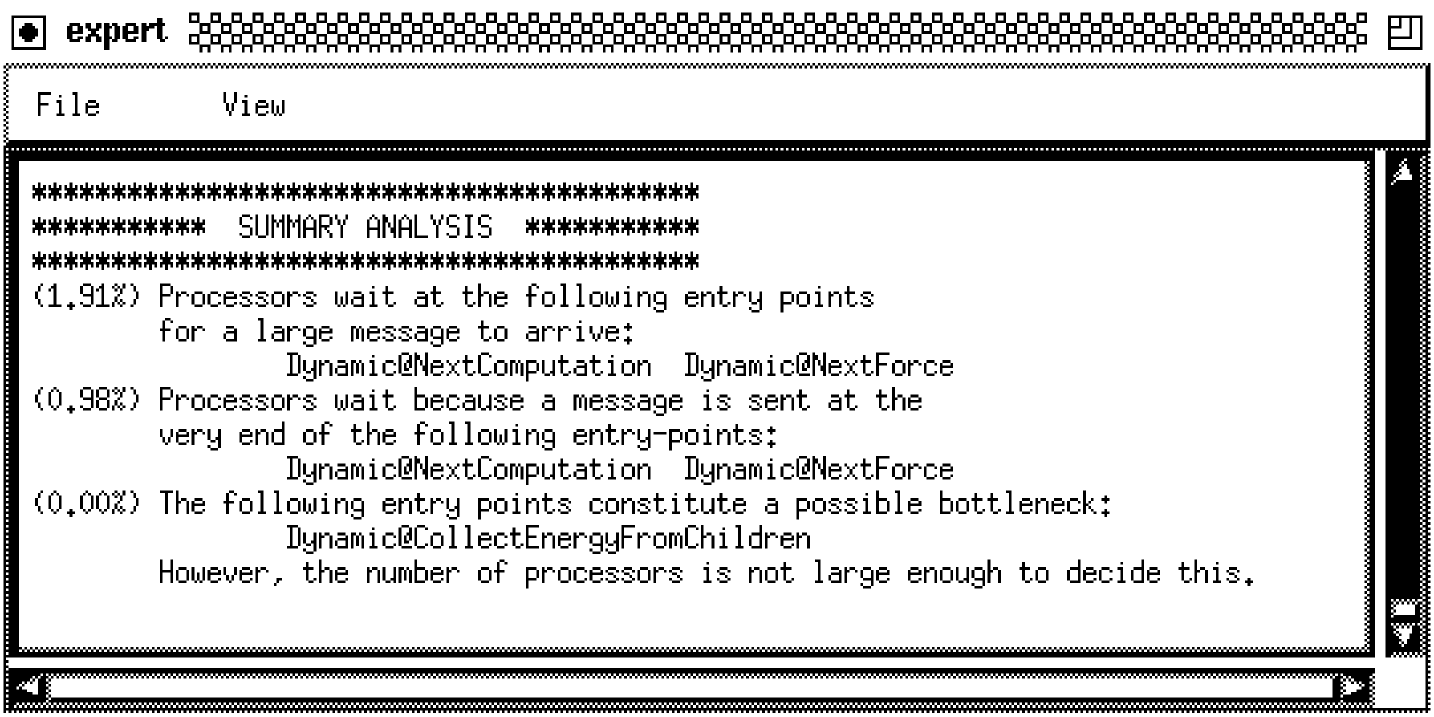
Case study: EGO



- *NextComputation* is the source of problem.
  - it computes Coulomb forces
  - forces must be added to message
  - since forces are not available till its completion, the entire message is held up until the end
- Solution?
  - coordinates do not change: send them out immediately
  - send a separate packet containing forces at the end

Case study: EGO

- Result: execution time reduced from 660s to 600s (9% improvement)
- New analysis



Conclusion

- Automatic performance analysis is feasible
  - preliminary version
- Automatic information about program behavior
  - through language constructs and system libraries for Charm
- What's needed for more advanced analysis?
  - more information
  - more techniques