Software for Parallel computing

Parallel computers : massively parallel, SMPs, workstation clusters.

Main goal : high performance

Parallel software development is difficult.

Parallel Software : Issues

- Decomposition
  - too large grainsize : less parallelism
  - too small grainsize : large overhead
- Mapping
  - load balance
  - communication locality
- Scheduling
  - critical path
- Machine-specific implementation

Problems : Performance and Programmability

Only experts can get good performance

- How to get better performance from parallel programs ?

Complex issues make parallel programming difficult

- How to make parallel software development easier ?

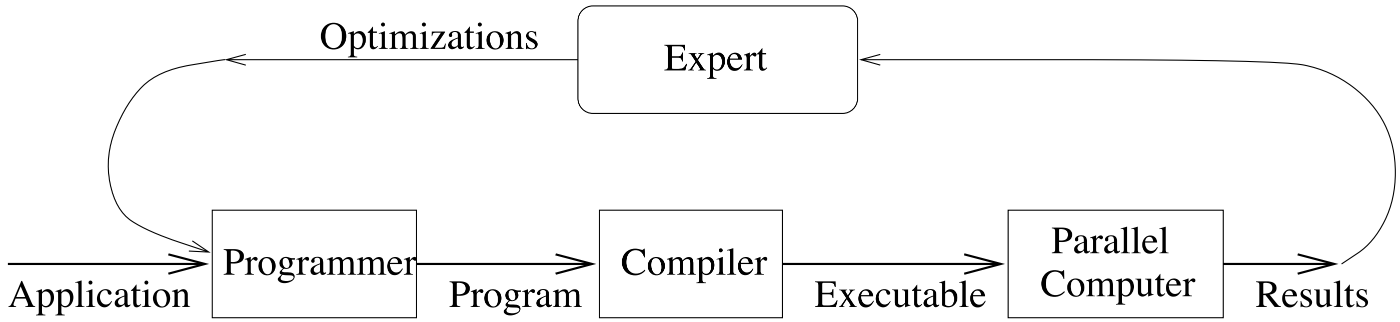
How to eat the cake and have it too ?!

Approach

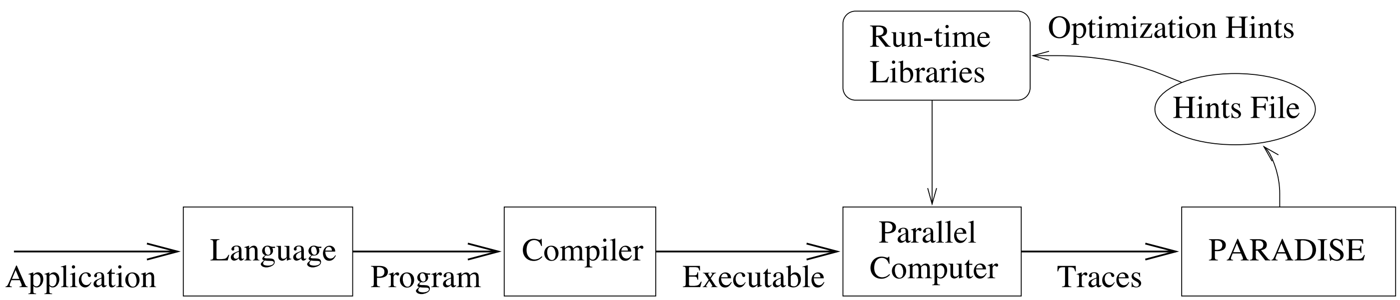
- use object-orientation
  - encapsulate complex details, make code reuse easier
  - objects naturally represent independent parallel computations
  - programmer only specifies decomposition into objects
  - system tries to automate everything else
- develop automated “expert” optimization tools
  - should embody the experience of good parallel programmers

Automated optimization tools

Typical parallel software development cycle :



- Parallel programming skills not widespread
- Need automated expert optimization tools



Program development with run-time optimizations driven by the Paradise post-mortem analysis tool

Relation to previous work

Performance analysis tools

- most existing tools visualize performance data
- a few tools (Projections, Poirot, Paradyn, MPP-Apprentice) diagnose performance problems
- our framework *solves* performance problems by automatic selection and incorporation of optimizations.

Compiler / runtime optimizations

- compiler optimizations alone are inadequate in many cases, need to be complemented by runtime optimizations
- existing runtime systems do not incorporate application-specific information / post-mortem analysis

- most automatic optimization research is for loop-based / data-parallel models.

Scope / breadth

- parallel object-oriented model allows dynamic creation of work, asynchronicity, irregularity, as opposed to data-parallel/SPMD models.
- Charm++ model places greater responsibility on runtime, thus more challenges and opportunities for automatic optimization.
- our framework automates optimizations for static and dynamic placement, scheduling, grainsize control and communication.

Charm++ : Overview

A parallel object-oriented language based on C++. Derives most of its features from the Charm parallel programming language.

Essential features :

- Parallel objects called *chares*
- Remote object creation, dynamic load balancing
- Asynchronous method invocations using global "object handles"
- Message-driven (actor-like) execution
- Parallel object arrays
- Prioritized scheduling

Why runtime optimization

Compiler optimizations alone are inadequate

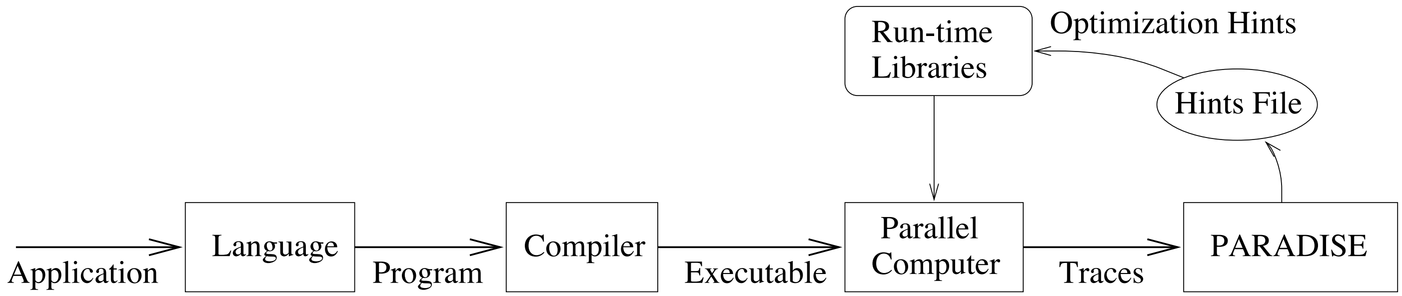
- Unpredictable parallel execution environment
- Unpredictable computational needs of applications
- Difficult to analyse C++-based parallel o-o languages
- Separate compilation reduces global/interprocedural information
- Many parallel programming environments are library based

Compiler optimizations must be complemented by runtime optimizations.

Why post-mortem analysis

- Program-specific information needed to parameterize and guide optimizations.
- Compilers cannot provide all the information required.
- Runtime analysis cannot detect global/spatial problem structure or make predictive decisions easily.
- Post-mortem analysis is anyway an integral part of manual development cycle.

Paradise : Automatic post-mortem analysis



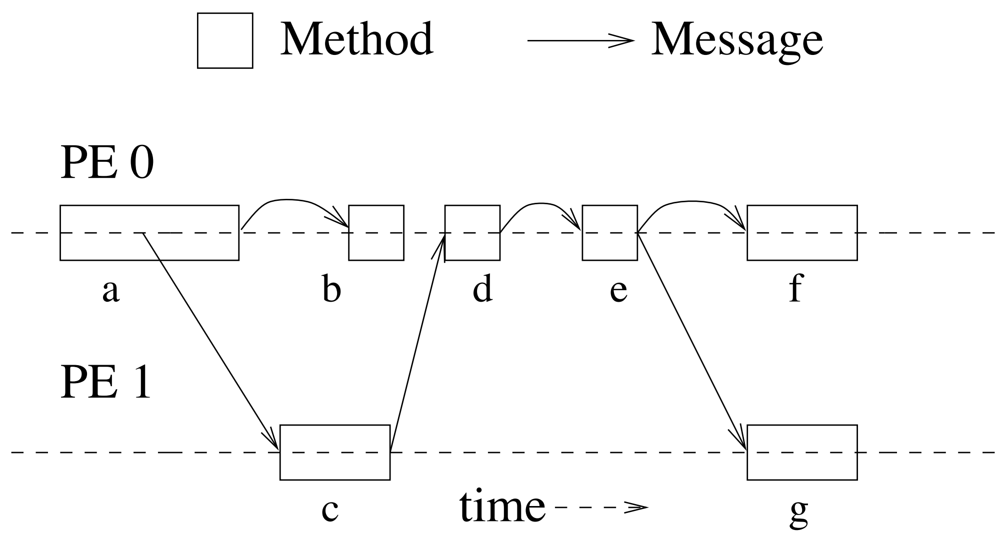
Program development with run-time optimizations driven by the Paradise post-mortem analysis tool

1. Compile program
2. Run program (generates traces of execution)
3. Run post-mortem analyzer tool (generates hints file)
4. Run program again : runtime libraries use hints to optimize execution

Program representation

Parallel program represented by event graph (dynamic task graph)

- original version designed for Projections tool.
- vertices = method invocations, edges = messages.
- add intra object dependence edges
- group method invocations by object instance



Non-determinism

Non-determinism affects analysis of program characteristics.

Causes :

- Inputs / Number of processors
- Adaptive placement
- Adaptive scheduling
- Adaptive granularity control
- Speculative execution

Solution : find application-level characteristics.

Handling non-determinism

Inputs :

- assume only size of computations change (most applications)
- generate only application-specific hints
- collect input-specific information at run-time

Adaptive scheduling, placement do not affect event graph.

Analyzing Optimizations

- identify / create control points where runtime libraries can affect program execution
- identify / design alternate optimization mechanisms to be applied at the control points
- develop strategies/heuristics to select between mechanisms
- identify program characteristics required to parameterize mechanisms / guide strategies
- develop techniques to automatically extract characteristics from event graph
- develop techniques to generate concise hints

Dynamic and static object placement, scheduling, granularity control, communication reduction.

Optimizing Dynamic Object Placement

Aims : Balance processor loads, and keep heavily interacting objects together

Control points : from seed creation through seed dispatch (object creation)

Schemes : Randomized, round-robin, neighbor averaging, centralized manager, hierarchical manager, etc.

Runtime information required : processor loads, load per object, interactions between objects

How to choose between the schemes ?

Heuristics for Dynamic Object Placement

```
if ( object creation is centralized )
    if ( all objects have the same grainsize )
        Choose round-robin
    else
        Choose hierarchical-manager
else
    if ( there is significant inter-object communication )
        Choose neighbor-averaging
    else if ( average grainsize is sufficiently large )
        Choose hierarchical-manager
    else if ( all objects have the same grainsize )
        Choose round-robin
    else
        Choose neighbor-averaging
```

Results for dynamic object placement

Program	Default	Automatic
Variable-Grainsize	2624	2290 (dist-mgr)
Heavy Communication	7685	6326 (nbr-avg)
Fibonacci (regular tree)	69	29 (tree)

Table 1: Time (in milliseconds) for different programs using dynamic object placement. (Tracing is off for all results, default mapping is round-robin).

Optimizing static object placement

Applies to multi-dimensional parallel object arrays

- No dynamic object creation
- Determine placement before computation begins

Aims : balance loads, reduce inter-processor communication

Use communication patterns, phase structure, grainsize patterns to determine best mapping of objects to processors.

Regular structure without phases

Regular : all array element objects have similar grainsizes, regular communication patterns (e.g. nearest neighbor)

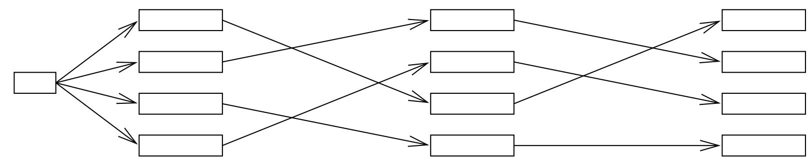
No phases : all objects active in all phases

Heuristic : Use block structured patterns

Find aspect ratio of block using amount of communication along dimensions. E.g. $\frac{Xsize}{Ysize} = \frac{Xcomm}{Ycomm}$

Regular programs with phases

Phases : all objects are not active in some phases.



Balance load within phases.

Generate load balance constraints between every pair of objects in every phase : the two objects should preferably not be assigned to the same processor.

Aim : satisfy as many constraints as possible.

For each mapping pattern (e.g. block-cyclic, multi-partition):
and for each set of constants in mapping expression: e.g.
$Map(i,j) = \frac{j}{a} MOD\ b + b * (\frac{i}{c} MOD\ d)$

- generate an assignment of objects to processors
- count the number of constraints satisfied
- choose the best pattern, constants

Irregular programs without phases

Significant variation in object grainsize or input-dependent load patterns.

E.g. irregular block-structured scientific applications.

Use run-time partitioning library : e.g. Orthogonal Recursive Bisection

Array-element objects must inherit from "load-array" class, and set load variable in constructor.

Partitioning starts at first synchronization point. Synchronous remapping of parallel object array follows partitioning.

Results for static object placement

Program	Default	Automatic
Jacobi	29.55	24.54 (block-block)
GaussElim (has phases)	34.90	34.90 (cyclic)
Irregular	7.94	2.51 (O.R.B.)

Table 2: Time (in seconds) for different programs using static object placement. (Default mapping is cyclic).

Scheduling Optimizations

Aim : Select order of execution of methods (messages) to minimize completion time.

Mechanism : prioritization

- assign a priority to every message
- scheduler maintains a priority queue of messages
- method corresponding to highest priority message is invoked

Paradise finds the program's critical path, and prioritizes messages along it.

Heuristics for Optimizing Scheduling

Determine if the program has a critical path.

- longest-path heuristic
- perform depth-first traversal of the event graph.

Find which message types lie on critical path.

- assign higher priority if a type occurs more often on critical path
- assign lower priority if more often on non-critical paths

Find which objects are on critical paths (e.g. array element objects)

- assign higher priority if the object occurs earlier on critical path
- use linear pattern expression to relate object-coordinate to priority (Priority = a * object-id + b)

Results for optimizing scheduling

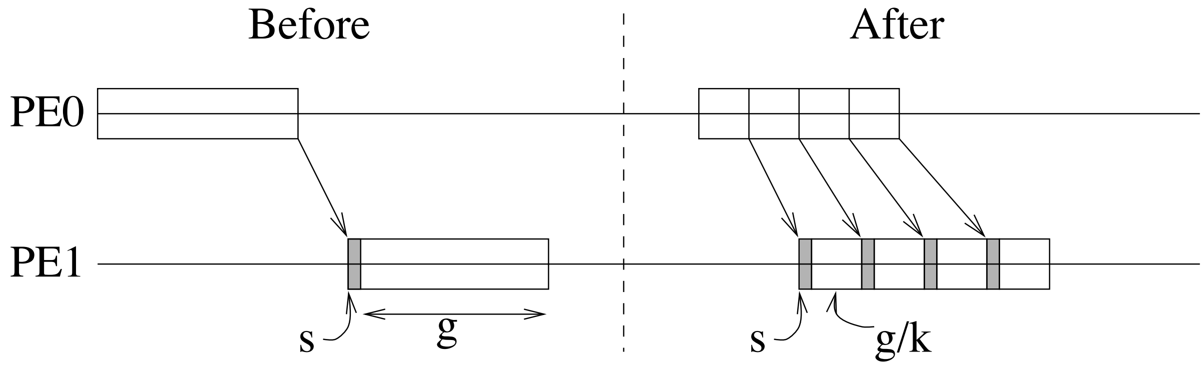
Program	Default	Automatic Priorities
GaussElim	43.58	34.90

Table 3: Time (in seconds) for Gauss Elimination with and without automatic prioritization.

Automating pipelining

Paradise finds a method on the critical path which executed after a long delay.

Degree of pipelining (formula from [Sinha95]): $k = \sqrt{\frac{\beta l + g}{s}}$



New control point needed for affecting pipeline degree.

Programmer obtains pipeline degree by calling "GetPipelineDegree()"

Automating message combining

Determine number of messages to combine, and sending/receiving processors.

Runtime uses this number as a hint, buffers messages, combines them at sender, and unpacks them at receiver.

E.g. special case : at synchronization points

- reduce messages from N to P
- find phases corresponding to synchronization points (pattern : $phasenum\%a + b = 0$) and enable combining for those phases.

Results for optimizing communication

Program	Before	After (Automatic)
Jacobi	50.57	24.54

Table 5: Time (in seconds) for Jacobi program before and after automatic message-combining.

Program	Manual (best)	Automatic
Poly-Overlay	15.09	15.37

Table 6: Time (in seconds) for Polygon-Overlay program with manual (optimal) and automatically pipelined versions.

Conclusion

Runtime optimizations improve parallel program performance, and they can be automated.

Future :

Parallel object-oriented programming (especially C++-based) is now main-stream.

Paradise, runtime optimization framework useful for simple parallel programs, but still more development needed:

- more optimization techniques
- better heuristics
- integration with compiler techniques