Compositional Development of Parallel Programs

Nasim Mahmood, Guosheng Deng, and James C. Browne
Overview

- Motivation
- Goals
- Programming Model
- Example
- Case Study
- Conclusions
- Current and Ongoing Work
- Future Directions
Motivation

- Optimization and adaptation of parallel programs is effort intensive
  - Different execution environments
  - Different problem instances
- Direct modification of complete application is effort intensive
- Maintenance and evolution of parallel programs is a complex task
Goals

- Order of magnitude productivity enhancement for program families
  - Develop parallel programs from sequential components
  - Reuse components
  - Enable development of program families from multiple versions of components
  - Automatic composition of parallel programs from components
Programming Model

- Component Development
  - Domain Analysis
  - Component Development
  - Encapsulate

- Program Instance Development
  - Analyze problem instance and execution environment
  - Identify attributes and attribute values
  - Identify data flow graph
  - Specify the program using the components and their interfaces
Component

Accepts interface
(profile, transaction, protocol)

Sequential Computation

Requests interface
(selector, transaction, protocol)
2D FFT Example

- Steps for 2D FFT computation
  - Partition given matrix row-wise
  - Apply 1D FFT to each row of the partition
  - Combine the partitions and transpose the matrix
  - Partition transposed matrix row-wise
  - Apply 1D FFT to each row of the partition
  - Combine the partitions and transpose the matrix
  - Transposed matrix is the 2D FFT of the original matrix
## 2D FFT Example (Cont’d)

<table>
<thead>
<tr>
<th>Fft_row</th>
<th>Gather_transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Domain: fft</td>
<td>a) Domain: matrix</td>
</tr>
<tr>
<td>b) Input: matrix</td>
<td>b) Function: gather</td>
</tr>
<tr>
<td>c) Element_type: complex</td>
<td>c) Element_type: complex</td>
</tr>
<tr>
<td>d) Algorithm: 1d-fft</td>
<td>d) Combine_by_row: true</td>
</tr>
<tr>
<td>e) Apply_per_row: true</td>
<td>e) Transpose: true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distribute</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Domain: matrix</td>
<td>a) Domain: print</td>
</tr>
<tr>
<td>b) Function: distribute</td>
<td>b) Input: matrix</td>
</tr>
<tr>
<td>c) Element_type: complex</td>
<td>c) Element_type: complex</td>
</tr>
<tr>
<td>d) Distribute_by_row: true</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2. Domain Analysis of the Components**
2D FFT Example (Cont’d)

Fig. 1. Data Flow Graph of 2D FFT Computation
Requests interface of Initialize

Accepts interface of Distribute

2D FFT Example (Cont’d)

selector:
  string domain == "matrix";
  string function == "distribute";
  string element_type == "complex";
  bool distribute_by_row == true;
transaction:
  int distribute(out mat2 grid_re,out mat2 grid_im, out int n,
                 out int m, out int p);
protocol: dataflow;

profile:
  string domain = "matrix";
  string function = "distribute";
  string element_type = "complex";
  bool distribute_by_row = true;
transaction:
  int distribute(in mat2 grid_re,in mat2 grid_im, in int n,
                 in int m, in int p);
protocol: dataflow;
Matching of
  - Selector and profile
  - Transactions
  - Profiles

Matching starts from the selector of the start component

Applied recursively to each matched components

Output is a generalized data flow graph as defined in CODE (Newton ’92)

Data flow graph is compiled to a parallel program for a specific architecture
2D FFT Example (Cont’d)

{selector:
    string domain == "fft";
    string input == "matrix";
    string element_type == "complex";
    string algorithm == "Cooley-Tukey";
    bool apply_per_row == true;
transaction:
    int fft_row(out mat2 out_grid_re[],out mat2
                out_grid_im[], out int n/p, out int m);
protocol: dataflow;
}index [ p ]

profile:
    string domain = "fft";
    string input = "matrix";
    string element_type = "complex";
    string algorithm = "Cooley-Tukey";
    bool apply_per_row = true;
transaction:
    int fft_row(in mat2 grid_re,in mat2 grid_im,in int n,
                in int m);
protocol: dataflow;
2D FFT Example (Cont’d)

selector:
  string domain == "matrix";
  string function == "gather";
  string element_type == "complex";
  bool combine_by_row == true;
  bool transpose == true;
transaction:
  int gather_transpose(out mat2 out_grid_re, out mat2 out_grid_im, out int me);
protocol: dataflow;
profile:
  string domain = "matrix";
  string function = "gather";
  string element_type = "complex";
  bool combine_by_row = true;
  bool transpose = true;
transaction:
  int get_no_of_p(in int n, in int m, in int p, in int state);
  int gather_transpose(in mat2 grid_re, in mat2 grid_im, in int inst);
protocol: dataflow;

Requests interface of FFT_Row

Accepts interface of GatherTranspose
selector:
    string domain == "matrix";
    string function == "distribute";
    string element_type == "complex";
    bool distribute_by_row == true;

transaction:
    %{ exec_no == 1 && gathered == p }%
    int distribute(out mat2 out_grid_re, out mat2 out_grid_im, out int m, out int n*p, out int p);

protocol: dataflow;
Fast Multipole in Short

- Compute the Coulomb Energy of point charges in linear time
- Transforming the information about a cluster of charge into a simpler representation which is used to compute the influence of the cluster on objects at large distances by scaling all particles into hierarchy of cubes in different levels
- Can be extended and applied to astrophysics, plasma physics, molecular dynamics, fluid dynamics, partial differential equations and numerical complex analysis
Generalized Fast Multipole Solver – Matrix Version

- Many generalized $N$-body problems can be treated as multiple FMM problems which share the same geometry. This feature can be exploited by combining the generalized charges into a vector.

- Generalized FMM is an extension of the FMM algorithm to multiple “charge types”.

- More efficient FMM translation routines could be built using BLAS routines.
FMM Domain Analysis

- **Six Translation Components**
  - Particle charge to Multipole (finest partitioning level)
  - Multipole to Multipole (between all partitioning levels, from the finest to the coarsest)
  - Multipole to Local (all partitioning levels)
  - Local to Local (between all partitioning levels)
  - Local to Particle potential and forces (finest partitioning level)
  - Direct Interaction (finest partitioning level)

- **Two Utility Components**
  - Distribute – Distribute Pre-Calculated Local Coefficient matrices according to Interaction list
  - Gather – Gather Local coefficients
Space-computation Tradeoffs For Matrix-structured Formulation of the FMM Algorithm

- Simultaneous computation of cell potentials for multiple charge types
- Use of optimized library routines for vector-matrix and matrix-matrix multiply
- Loop interchange over the two outer loops to improve locality
Flow Graph for Sequential FMM
Flow Graph for The Parallel Version

Initialization

P2M
M2M
M2L
Distribute

L2L
Distribute

L2P
Direct

Collect

L2L
Collect

L2P
Direct

Terminate

P[0]

P2M
M2M
M2L
Distribute

Direct

P[1]
Sequential Running Time - New/Old WRT number of charge types
Conclusions

- Effort in domain analysis is not trivial
- Suitable for
  - Large applications that are to be optimized for several different execution environments
  - Large applications that are expected to evolve over a substantial period of time
  - Large applications with multiple instances
- Competitive program performance
Current and Ongoing Work

- Implement evolutionary performance models of programs through composition of components
  - Abstract components
  - Concrete components
  - Performance model for specific architecture
- Componentize hp-adaptive finite element code and Method of Lines (MOL) code
Future Directions

- Combine with dynamic linking runtime system based on associative interfaces [Kane ’02]
- Implement more powerful precedence and sequencing operators for state machine specifications
- Integrate with Broadway [Guyer/Lin ’99] annotational compiler to overcome “many components” problem