The Higher-order Parallel Programming (HOPP) Model

Roopa Rangaswami
Department of Computer Science
University of Edinburgh, Scotland
ror@dcs.ed.ac.uk

Dagstuhl Seminar

Introduction - What is HOPP?

- Model of parallel programming based on BMF
- A program in HOPP is expressed as a composition of higher-order functions borrowed from BMF (*recognised functions*)
- Implementation attempts to exploit the inherent parallelism in recognised functions
- Emphasis on static analysis of programs to select cost-effective parallel implementations
- Applicable for *regular* problems

Features of the Model

- The Program Model - describes characteristics of a HOPP program
- The Machine Model - distributed-memory; abstract from physical topology by using MPI to handle interprocessor communications
- Cost Model - computes the costs of all possible parallel implementations of a given HOPP program and selects a cost-effective one

The Program Model

- Independent of base language - strict functional

- Example of a program in HOPP

  prog xs = (**map fold** plus 0 $\circ$ **map map** sqr) xs;

- Each component - *phase*; program has one or more phases

- A phase has nested instantiations of recognised and user-defined functions

- *Only* allowable program format

- Recognised functions could be implemented in parallel; user-defined functions implemented sequentially

The Cost Model

- Cost analysis carried out at compile-time
- Implemented in the form of an *analyser*
- Parallelism exploited within each phase; phases themselves sequential
- Number of possible parallel implementations for a phase depends on number of nested recognised functions in it
- No recognised functions in phase? $\Rightarrow$ sequential implementation

The HOPP Code Generator (Joint work with Tristan Tarrant)

- The output of the analyser is a specification of the parallel implementation strategy for each phase of the program
- Output of analyser is input to code generator which automatically generates parallel code
- A code generator which produces C++ SPMD code using MPI for communication has been designed and implemented
- Why C++?
  - Efficiency - optimisation strategies such as loop unrolling and function in-lining
  - Portability - C++ compilers exist for virtually every machine
  - Object-oriented - New datatypes can be defined and handled as if they were native ones

Main Features of the Code Generator

- Can handle user-defined types
- Built-in type inference
- Customised implementation of list class to efficiently handle *nested*, *polymorphic*, *distributed* lists
- List class based around a structure of *dynamic arrays* - *dimension*, *limit*, *delta*
- Primitives for collective communication of lists (e.g. scatter, broadcast, gather) have been defined (MPI)
- If input and output variables of a phase are of identical type and size, data is modified *in place* - reduces overhead

Two Example Programs

- *Matrix Multiplication* and *3-D Transformations* were coded in the HOPP style

- Code was generated for predicted cost-effective parallel implementations and executed on the Cray-3TD

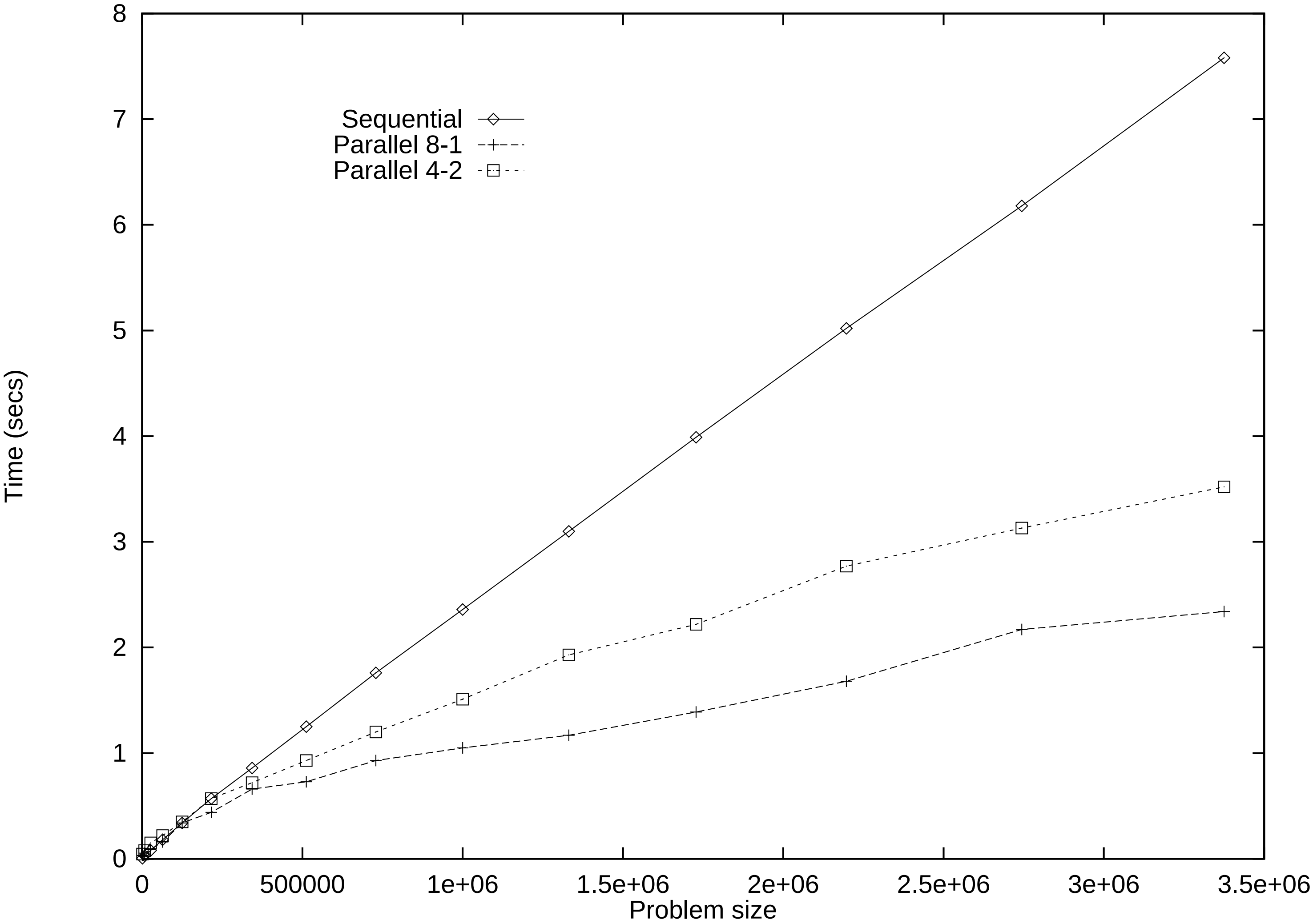
- Matrix Multiplication - Multiply two matrices A and B

  fun matmult times plus $A\ B^T =$
  (**map map fold** plus 0 $\circ$
  r_cross_product map2 times) $A\ B^T$

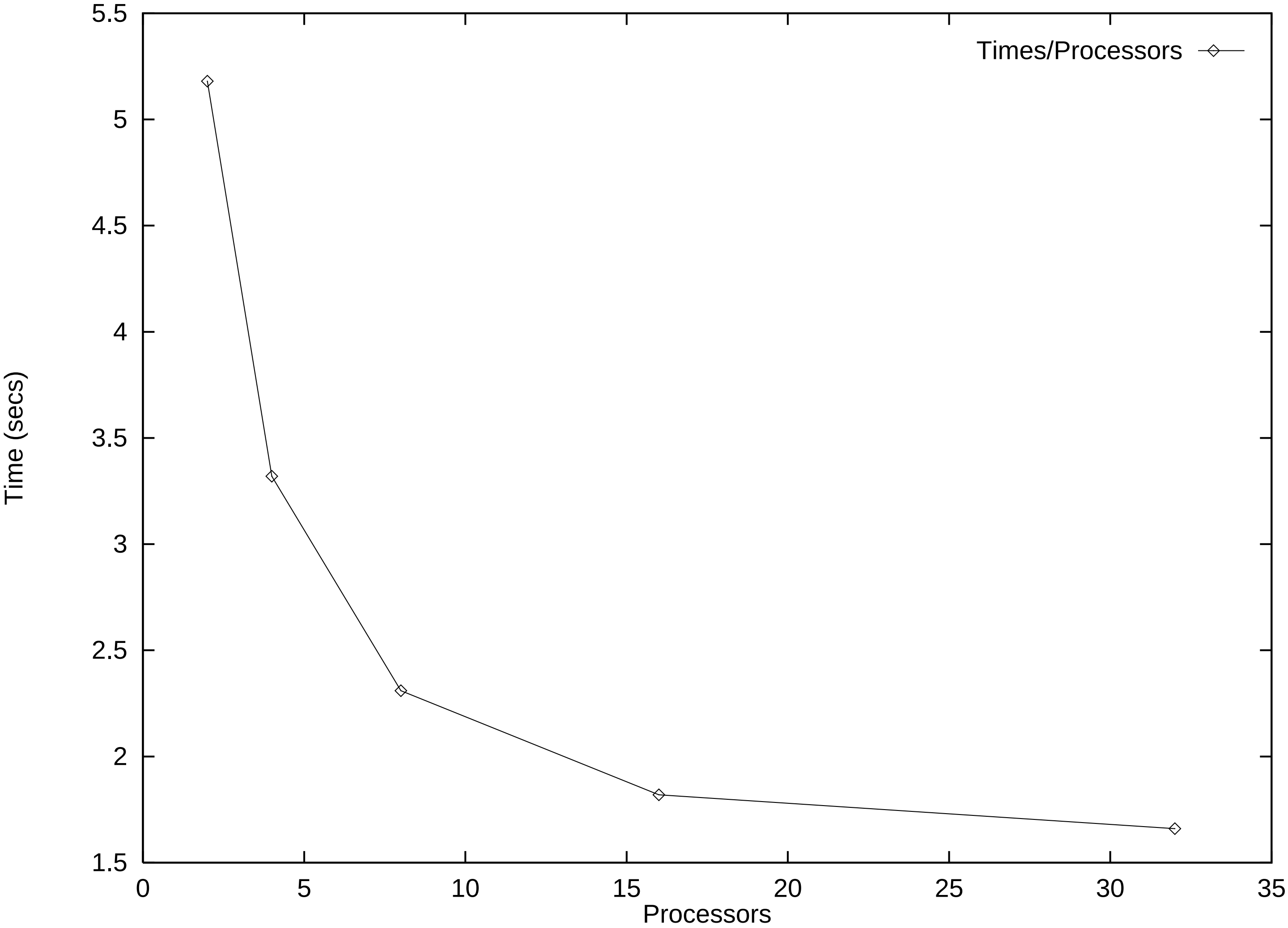
- 3-D Transformations - Input a list of 3-D points; translate, rotate, scale, clip and output list

  fun threeD trans rot scale clip $A =$
  (**filter** clip $\circ$ **map** scale $\circ$
  map rot $\circ$ **map** trans) A

Matrix Multiplication Results
(Sequential and Parallel Implementations)



Matrix Multiplication Results
(Changing the number of processors)

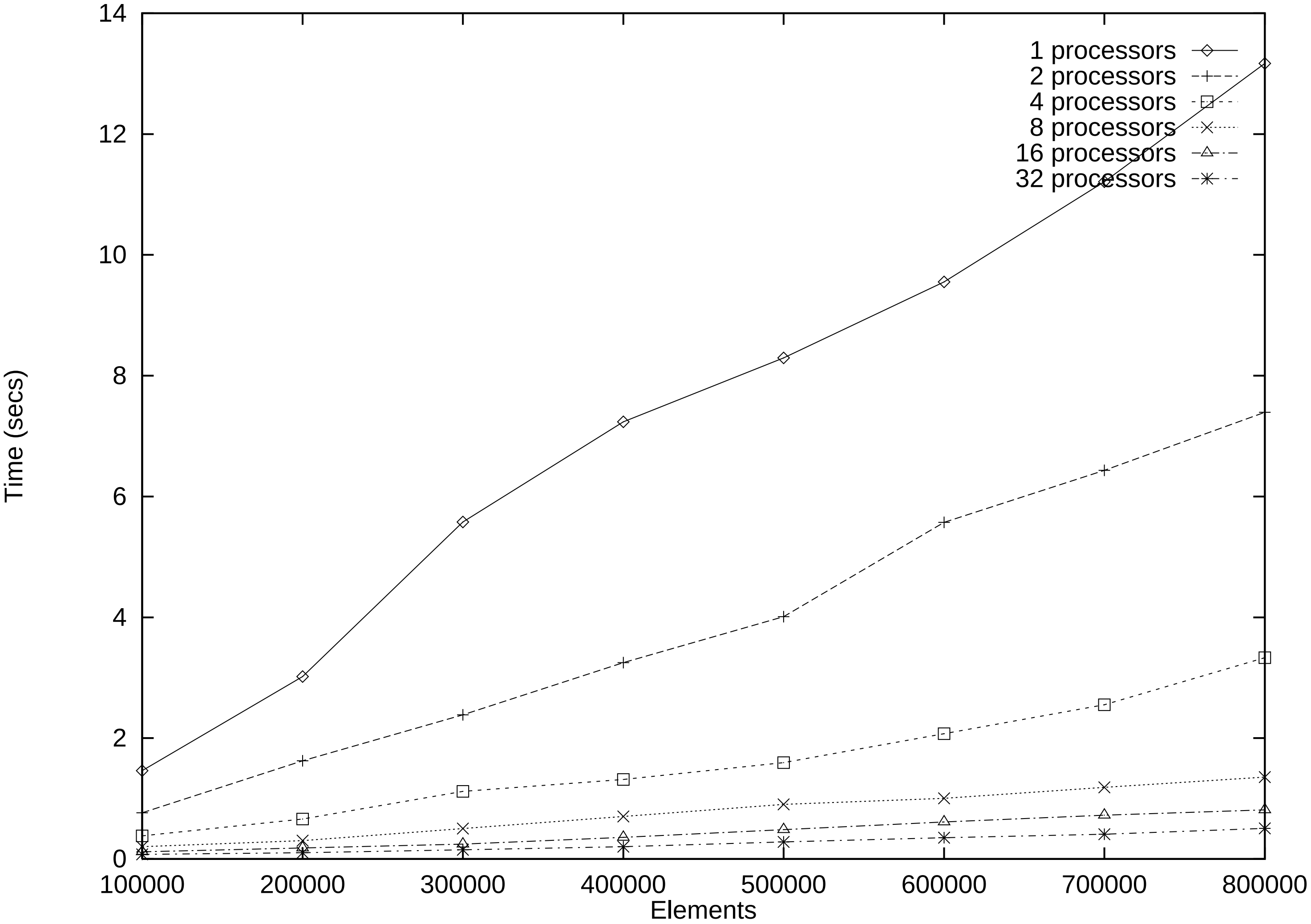


Analysis of Matrix Multiplication Results

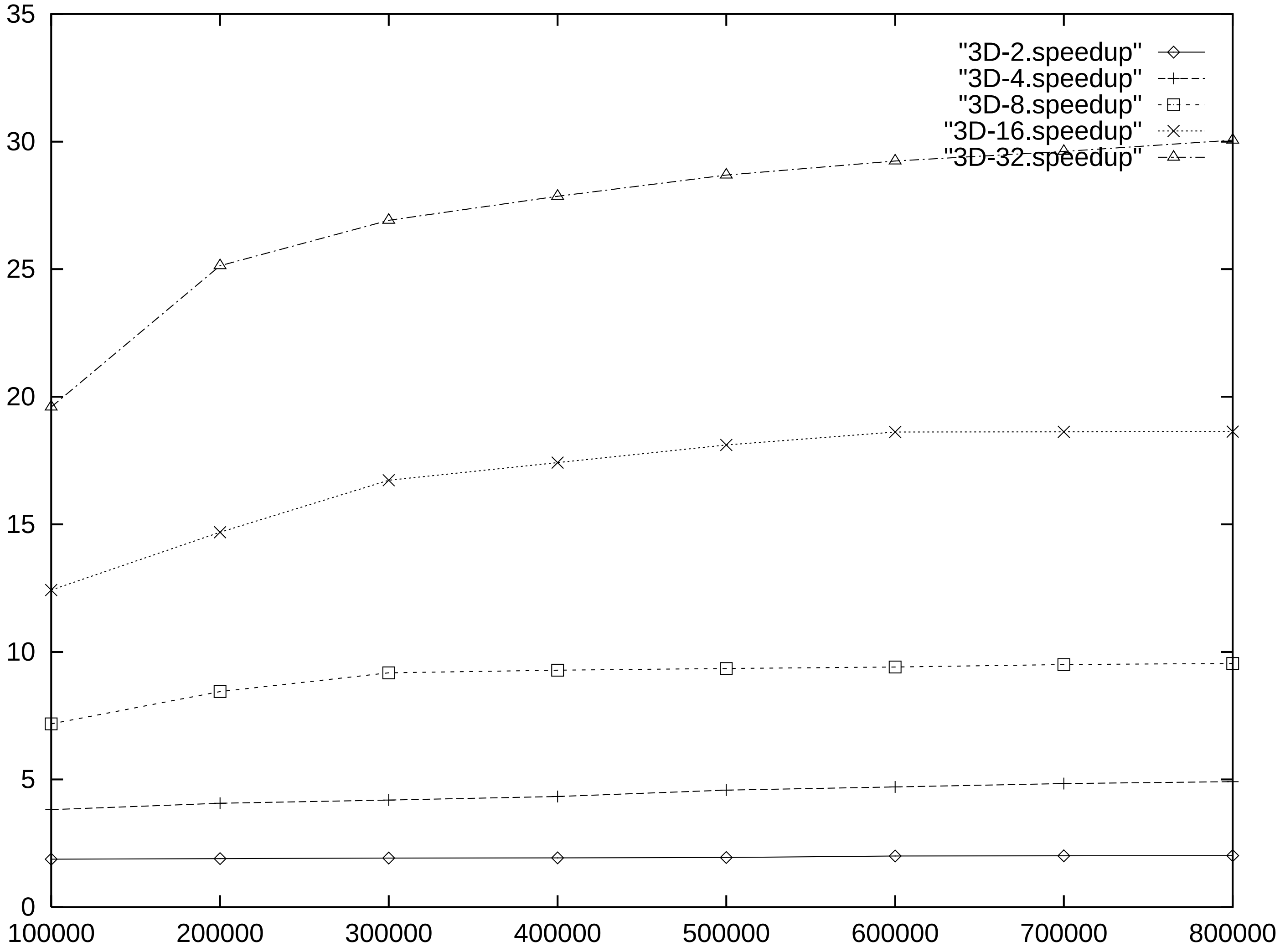
- Initially total number of processors was set to 8
- Two parallel implementations were considered
- Parallel implementation where only outer function is executed in parallel performs best
- Speedup levels off with increasing numbers of processors
  - Each processor has smaller amounts of data to process
  - Data distribution overhead > granularity of computation

3-D Transformation Results

(Input Size vs. Time)



3-D Transformation Results
(Speedup)



Analysis of 3-D Transformation Results

- Non-linear increase of curves caused by extra overhead of list processing

- More evident in case of lower numbers of processors since local lists are larger

- Resizing a list is expensive - number of such occurrences can be controlled to some extent by choosing *good* values for *limit* and *delta*

- Increasing the number of processors gives considerable speedup - mainly due to reduced overhead of list processing and low communication overhead

Expressivity of HOPP Model

- Results of the two simple examples tested on code generator are encouraging
- Practicality of model depends on its expressivity
- Work currently in progress involves testing expressivity by attempting to code a number of well-known problems in parallel programming in the style of HOPP
- These problems will then be parallelised using the HOPP scheme and executed using the code generator to test the performance of parallel implementations
- Enable the identification of classes of problems on which the model performs poorly
- ?