A Framework for an Automatic Hybrid MPI+OpenMP code generation

Gustavo Ramirez

March 31, 2017

Contents

OVERVIEW

Paper

A Framework for an Automatic Hybrid MPI+OpenMP code generation

Khaled Hamidouche, Joel Falcou, Daniel Etiemble
Laboratoire de Recherche en Informatique
University Paris-Sud XI, Orsay, France
hamidou,joel.falcou,de@lri.fr

Keywords: Generic Programming, Bulk Synchronous Parallelism, Performance Prediction, OpenMP, MPI, MPI+OpenMP

- OpenMP parallelization efficiency: using critical section primitives, the overhead of OpenMP threads management and false sharing can reduce the performances.
- MPI and OpenMP interaction: load balancing issues and

General description

Motivation:

- writing good hybrid code implies **expertise** and **performance analysis**
- hybrid code can be inefficient (MPI comms overhead, OpenMP critical sects, MPI+OpenMP interaction (e.g. load balancing, idle threads), etc...)
- $\Rightarrow$ the motivation is not dealing with all that

Goal (of this framework): produce efficient hybrid code for multi-core architectures from the source of a sequential function.

BACKGROUND

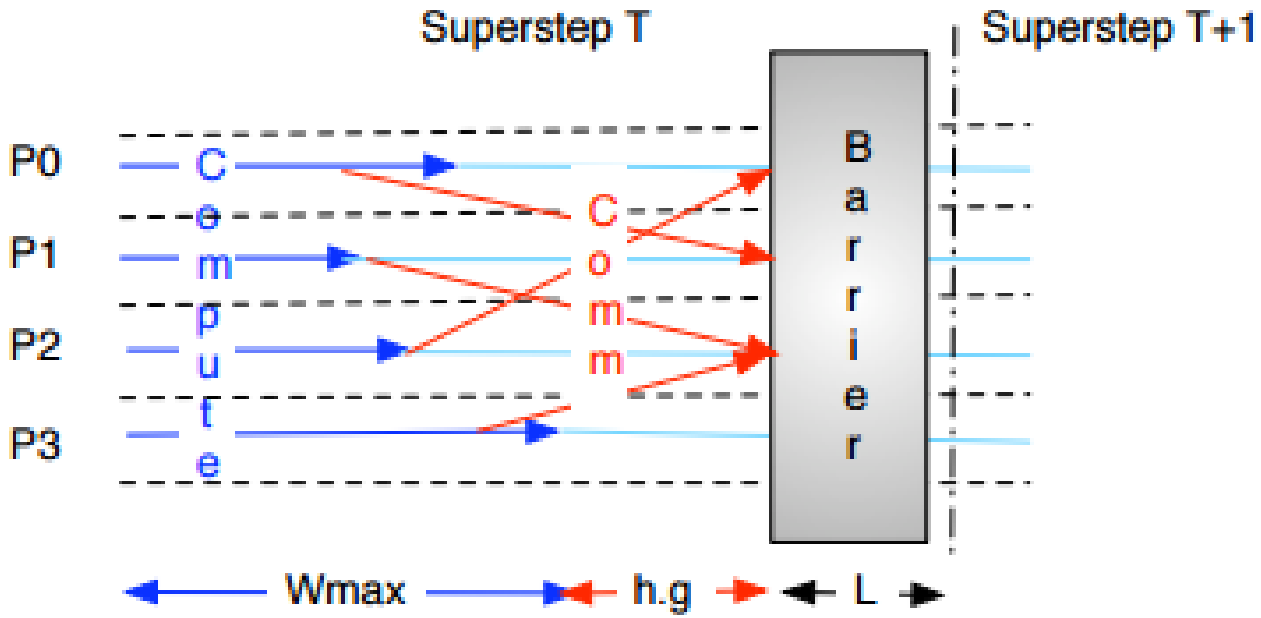
The BSP model

BSP model = bulk synchronous parallel model

The model is subdivided in:

- a machine model:
  - description: set of processors linked through a comm medium supporting point-to-point comms and syncs
  - params: P (nr processors), r (speed, [r] = FLOPS), g (comm speed), L (sync duration)
- a programming model: (see next slide)
- a cost model: $\delta = W_{max} + hg + L$

The BSP model



h: maximal amount of data that is sent or received by one processor

The BSP model

Cons:

- cost of global syncs; can become dominant for large parallel machines
- to reduce the previous problem, use OpenMP to minimize sync overhead

Pros:

- simple analytical model; the BSP cost model facilitates the execution time estimation
- performance improvement; "best" number of processors and threads

SPMD

Single Program, Multiple Data.

Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.

Using other mode different from SPMD might imply high thread creation overhead, and serial sections may become bottlenecks.

SPMD

Pros:

- performance prediction is easier (split into Computation and Comm times)
- avoid false sharing
- load balancing is good, as all threads execute the same program

IMPLEMENTATION

Implementation specs

Framework specs:

- uses BSP cost model to estimate exec time
- finds the best configuration (nr MPI processes + OpenMP threads)
- generates hybrid code

Inputs: user source file containing seq code + XML describing parallel code
Outputs: hybrid MPI+OpenMP program

Implementation stages

Stages:

- analyzer: generates a numerical formula for exec time (with data size as param)
- searcher: uses previous formula and info from the "system profile" to estimate cost for all possible configs
- generator: uses the chosen configuration and generates the corresponding hybrid code

System profile: retrieves exec time for basic operations, obtained either from the processor's manual and design specs, or by benchmarking

Analyzer: computation model

Used to estimate T_{comm}. Counts the number of cycles needed to execute a sequential function.
Clang (with all optimizations: -O3) $\rightarrow$ LLVM (which counts the number of cycles in the bytecode).

Subdivided:

- processor model (no cache misses): $total = \sum_i C_i$, with C_i the hardware cost of each operation (e.g. load, store, etc.) i
- cache model (for cache misses): $cost = \sum_i^{levels}(M_i \cdot PEN_i)$, with the penalty PEN_i given in nr of cycles, and (j represents instructions):

$$M_i = \frac{1}{CAPACITY_i} \sum_j (nr.refs_i)(\beta_i)$$

Analyzer: communication model

- $T_{comm,k} = h_k \cdot g_k(P_k) + L_k(P_k),\ k \in \{MPI, OpenMP\}$
- System profile:
  - combines many benchmarks and sphinx to find $g_k(P_k)$ and $L_k(P_k)$
  - sphinx: tool for running performance tests of MPI, Pthreads and OpenMP
- $T_{comm} = \sum_k T_{comm,k}$

Searcher: goal

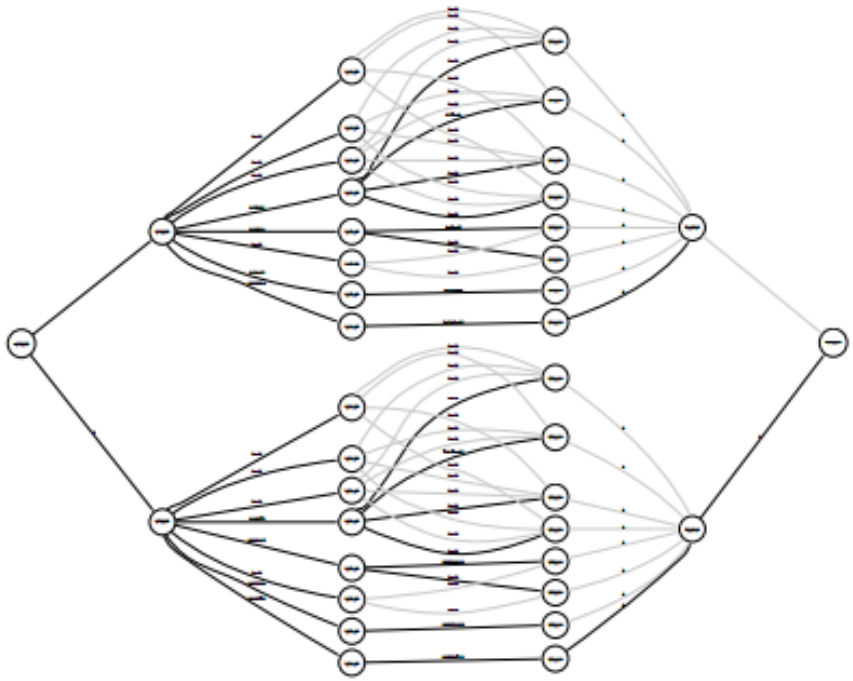
Uses the XML and the data from the analyzer.

Find the best combination number of MPI processes and OpenMP threads.

Factors:

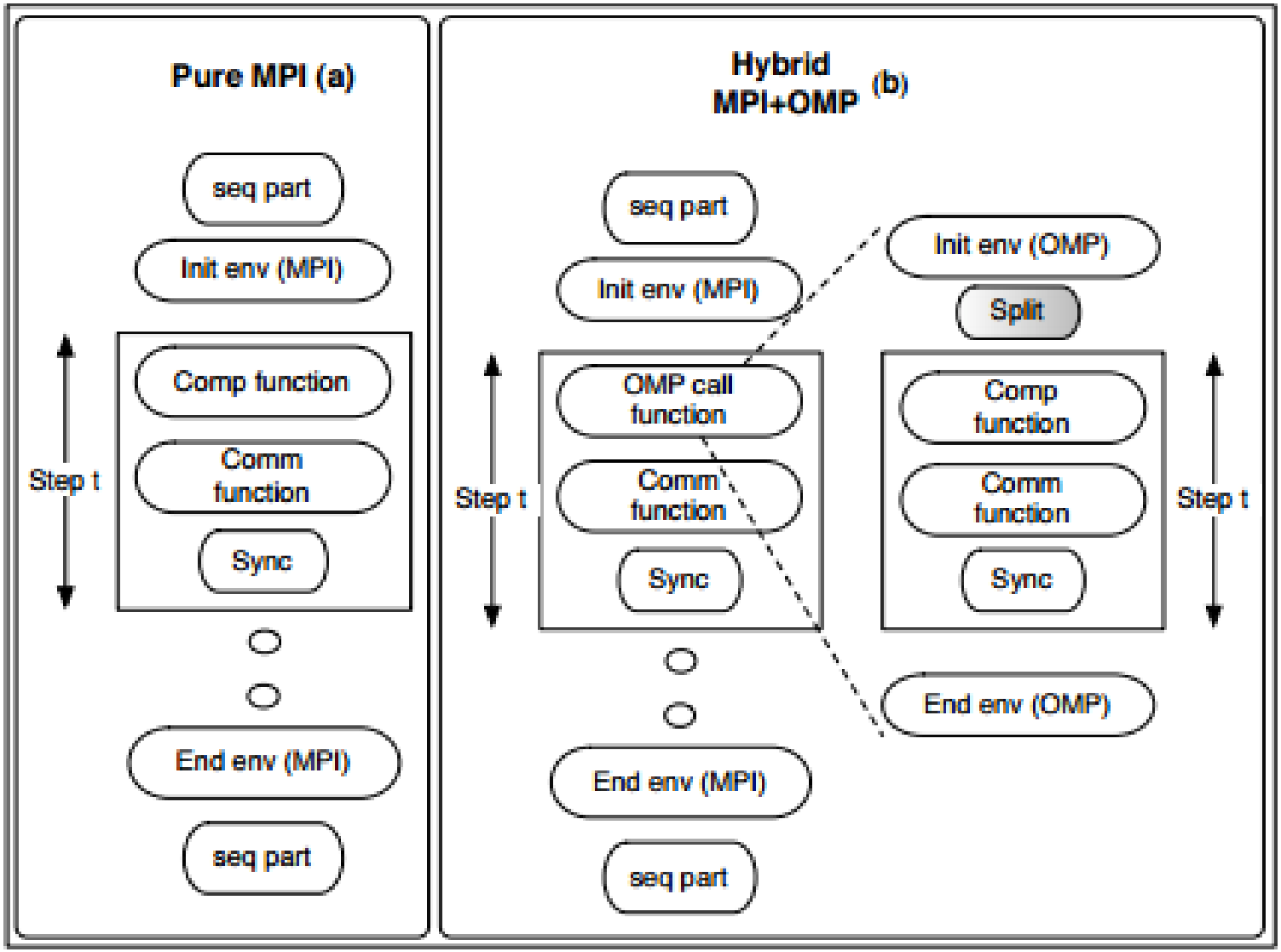
- number of MPI processes
- number of threads per process
- data size of the application

Searcher: graph

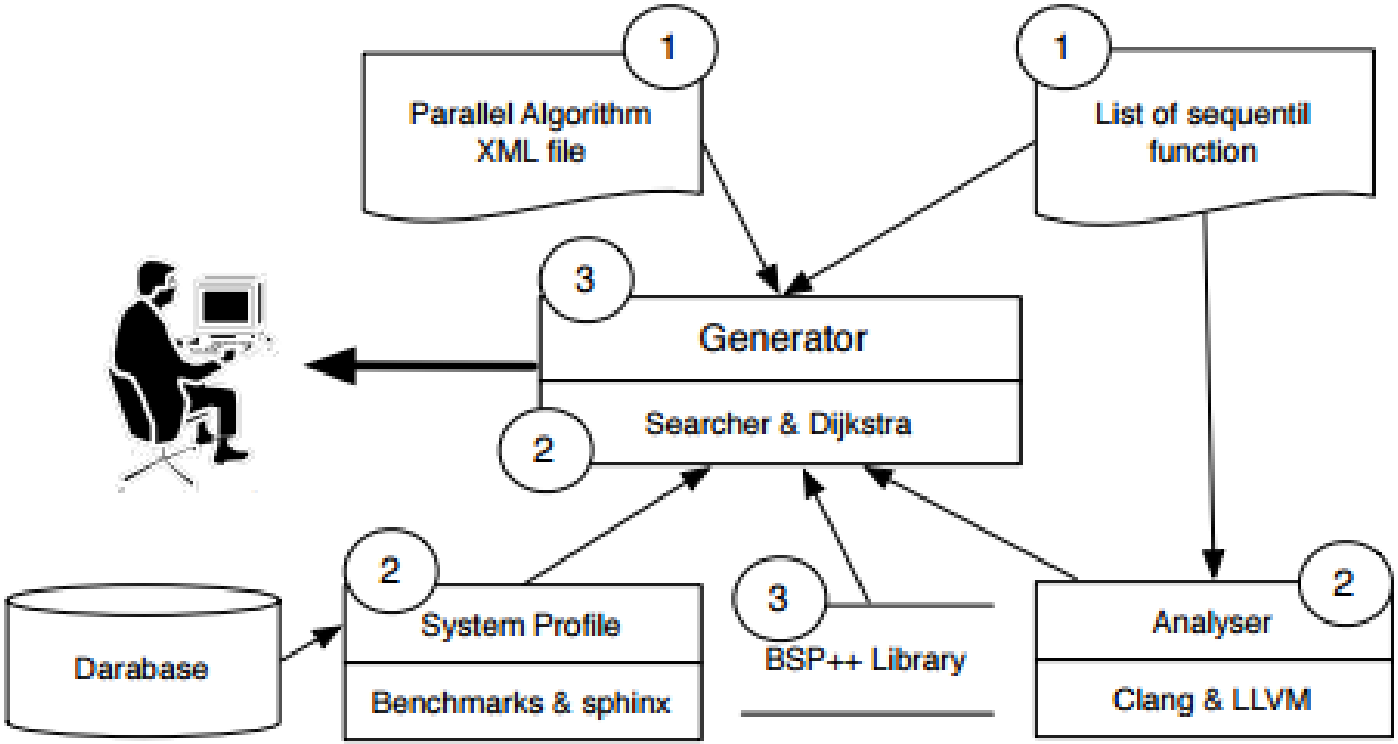


Vertex: $V(S_i, n, m, o)$. Edge: $V(S_i, n, m, o) \rightarrow V'(S_{i+1}, n, m, o')$.
Use of Dijkstra's shortest path algorithm.

Generator: BSP++ → hybrid structure



Global view


1
Parallel Algorithm
XML file
1
List of sequentil
function
3
Generator
2
Searcher & Dijkstra
Darabase
2
System Profile
Benchmarks & sphinx
3
BSP++ Library
2
Analyser
Clang & LLVM

Results: benchmarks and hardware

Four benchmarks:

- inner product
- vector-matrix multiplication
- matrix-matrix multiplication
- Parallel Sorting by Regular Sampling algorithm

Hardware:
4 nodes. Each node is a bi-processor bi-core 2.6 GHz, 4 GB RAM, 2 MB of L2 cache.

Results: model performance

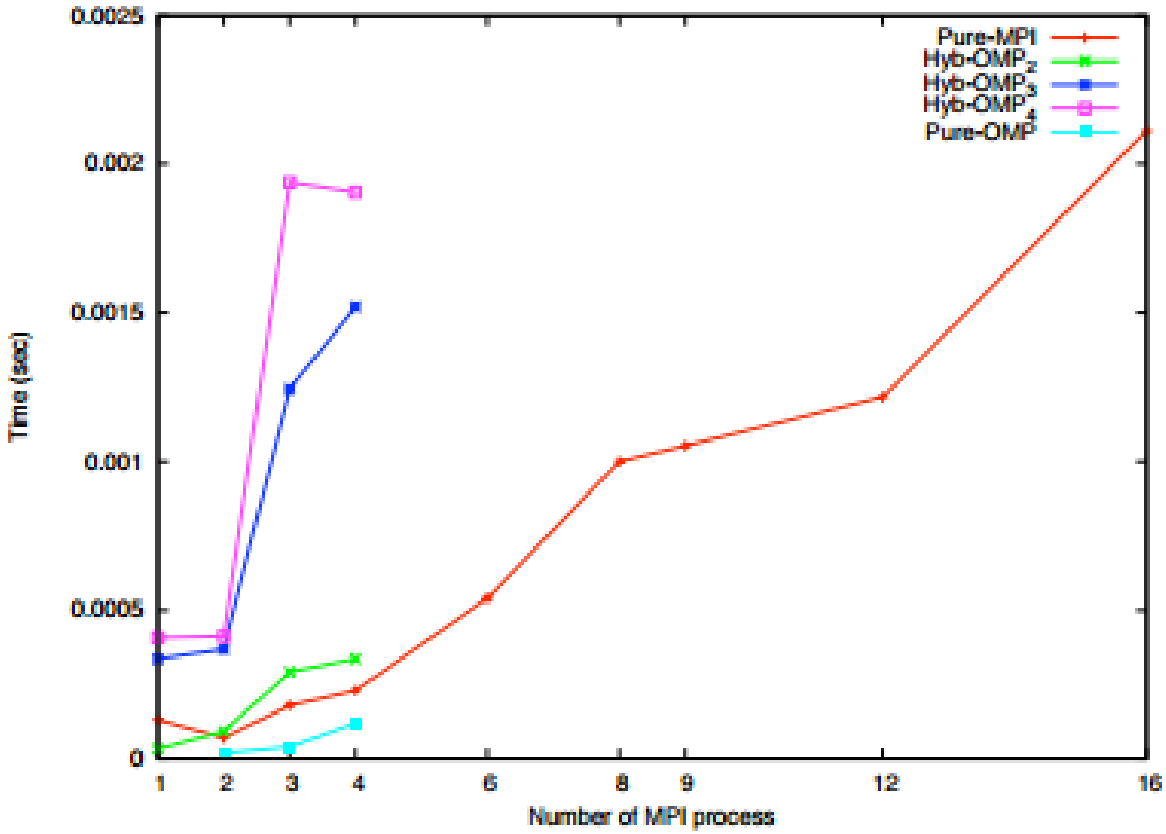


Figure 5. Execution time for all possible configurations for the inner product program with a small data size (64 k).

Results: model performance

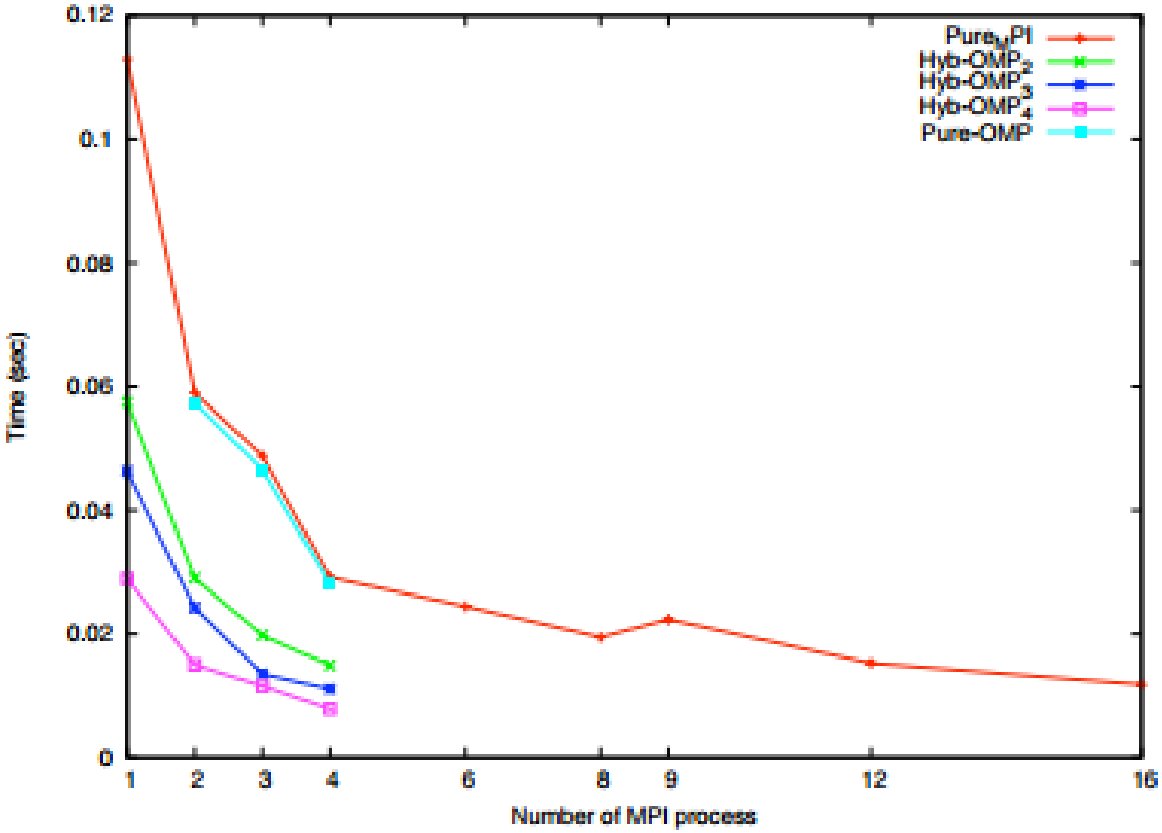


Figure 6. Execution time for all possible configurations for the inner product program with a large data size (128 M).

References

HAMIDOUCHE, K., FALCOU, J., AND ETIEMBLE, D. *A Framework for an Automatic Hybrid MPI+OpenMP code generation*. Proceedings of the 19th High Performance Computing Symposia (HPC '11) (2011)

Thank you