Automatic Program Parallelization

Jan Kwiatkowski
Institute of Informatics
Wroclaw University of Technology
Poland
Introduction

• Nowadays multicore processors become more and more popular for both commercial and home usage
• Developing parallel programs is more complicated than serial
• Global tendency: Put more cores instead of increasing clock frequency
• Program execution has to maximize processor usage
• Dedicated solution causes that program is not portable what means that during program execution on the processor with different number of processing units, the efficiency can be worse
• As an alternative solution the automatic program parallelization can be used
Different approaches

• Mostly with programmer „hints” what can be parallelized, e.g.:
  o **Open MP** – standard which is used by many solutions
  o **RapidMind** - execution environment which uses own virtual machine for code execution
  o **Octopiler** - is automatic SIMD’ization of code and automatic user-directed exploitation of shared memory parallelism – used Open MP

• Hardware parallelization:
  • **SUIF** - speculative thread level parallelism
Dependencies in the program

Data dependence

It is a partial order or precedence relation on the statements of a program. Statement T depends on statement S, denoted $S \rightarrow T$, if there exists an instance $S'$ of S, an instance $T'$ of T, and a memory location M, such that:

1. both $S'$ and $T'$ reference M, and at least one reference is a writer;
2. in sequential execution of the program, $S'$ is executed before $T'$;
3. in the same execution, M is not written between the times $S'$ finishes and the time $T'$ starts.

There are three types of dependencies based on the references to M:

- Flow dependence
- Anti dependence
- Output dependence
Control dependence

Determines the ordering of an instruction, $i$, with respect to a branch instruction so that the instruction $i$ is executed in correct program order and only when it should be.
Parallelization process

1. Performing a **dependence test** to detect potential program parallelism

2. Restructuring the program into a few blocks which can be executed in parallel. During this step different program transformations are used to obtain the highest level of parallelism in a program

3. Generating **parallel code for a target architecture** by scheduling program blocks on available processing units and for available operating system
Tool Architecture 1/4

- Serial code
  - Code Analyzer
  - Optimizer
  - Code generator
    - Parallel code
- Compiled code
  - Compiler
  - Utterly platform dependent
- Platform independent
  - Platform dependent for system calls only
• Code Analyzer creates program representation in the memory as a **tree of recognized program blocks**

• Optimizer performs the following steps:
  – Looks for internal parallelism within the program blocks and performs loop analysis
  – Creates **dependency graph** for the whole program as well as for each program block
  – Analyses dependency graph and checks for possible transformations
  – Determines the most suitable (optimal) one. It is based on information about target platform (number of processing units) and with estimated execution time for each possible transformation set

• Code Generator generates final code for target platform (depending on OS and system architecture).
Determining the most suitable transformation set (for loops) is performed in the following steps:

- Checking possibility of using „the best” transformations from statistical point of view:
  - Loop Splitting
  - Loop Distribution
- If **possible** – checking other transformations corelated with previous one – trying it in different order
- If **impossible** – checking other transformations to determine whether program can be restructured to use one of „the best” transformation e.g. Loop Interchange for nested loops
- Estimating time of each possibility
- Choosing most efficiency transformations set
Parallelization is achieved by using threads:
- Because threads handling is different in each OS a proper "driver" for each system has to be implemented
- In fact all system calls has to be included in the "driver"

Block parallelization (e.g. loop) is achieved in the following steps:
- Creating proper data structures for each thread
- Replacing parallelized block with threads calls
- **Making necessary synchronization and communication**
- Sending data for each thread
- Gathering results to the main thread
Available loop transformation

- **Loop splitting** – divides the loop into parts with different indexes with same loop body
- **Loop distribution** - divides the loop into parts with the same indexes but each loop uses different parts of the original loop
- **Loop unrolling** - restructures a loop by writing the iterations as straight-line code segments
- **Loop reversal** – the new loop is reversed version of original one
- **Loop interchange** – changes the order of execution of nested loops
Some results of experiments - Loop splitting

The tests were executed on the Intel Xeon 1,86 GHz (2 core), Intel Core 2 Quad Q6600 2,4 GHz (4 core)

![Graph showing speedup vs number of cores]

- Real speedup
- Estimated speedup
Some results of experiments - Loop splitting

100 iterations

- Real speedup
- Estimated speedup

Number of cores

Speedup

0,5
1
1,5
2
2,5
3
3,5
4
4,5

1
1
1
3,86
Some results of experiments - distribution

100 iterations

Graph showing the speedup of 100 iterations with different numbers of cores.

- **Speedup** on the y-axis.
- **Number of cores** on the x-axis.

Lines and markers indicate the real and estimated speedup for 1, 2, and 4 cores.

- **Real speedup** (orange line with markers).
- **Estimated speedup** (purple line with markers).

Markers at:
- 1 core: 1.0, 2.0, 2.6
- 2 cores: 2.0, 1.7
- 4 cores: 4.0
Parallelization cost

Parallelization and compilation time

- Parallelization time
- Serial program compilation time
- Parallel program compilation time
- Parallelization and compilation time

Number of code lines

Time [ms]
Conclusions and future works

- The project is being studied, therefore currently it covers only a part of target functionalities.
- The prototype still misses a lot of features and not all types of instructions can be parallelized, they will be implemented in the further versions of the tool.
- Experiments performed using the first prototype indicates that the presented tool will be useful for many types of computers with multiple processing units.
- Speedup for each program is between 1 and number of processing units (so it is always good to use this tool).
- No specific knowledge about the used processors and experience in the parallel programming are required to build the parallel application.
Questions?