Parallelization of C programs through dependency analysis

Jos van Eijndhoven
jos@vectorfabrics.com

June 20, 2012

HPDC’12
Delft, The Netherlands
Discovering potential concurrency

PAREON

Sequential program

Parallel program

Parallel APIs
- OpenCL
- POSIX Threads
- OpenMP

Platforms
- Intel
- AMD
- ARM
- XILINX

Slow application and Silicon goes unused

Fast application Silicon put to work
Multi-processors will further expand

Partitioning workload across multiple cores:

- Data partitioning
- Functional partitioning

Tooling for dependency analysis

Example: LAMMPS

Conclusion
Programming parallel computers

- Multi-processor machines are all around us, ranging from mobile to super-computers.
- Multi-processor architectures are driven by technology factors: increased integration densities and power efficiency requirements.
- Parallel (multi-threaded) programs are required to exploit their capability.
- However, parallel programming is difficult and error-prone. Unfortunately, software productivity already is a major bottleneck.
- Distributed-memory architectures and heterogeneous processors (GP-GPUs) add further complications.

Application programmers need more help!
Creating multi-threaded concurrency

Basic fork-join pattern, created through different higher-level programming constructs

Main program thread

Concurrent computation threads

Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors.
Parallelization – two partitioning options

Source code:

```c
for (i=0; i<4; i++) {
    A(i);
    B(i);
    C(i);
}
```

Sequential execution order:
- A(0) A(1) A(2) A(3)
- B(0) B(1) B(2) B(3)
- C(0) C(1) C(2) C(3)

Data partitioning:

- Fork
- A(0) A(1) A(2) A(3)
- B(0) B(1) B(2) B(3)
- C(0) C(1) C(2) C(3)
- Join

Functional partitioning:

- Fork
- A(0) A(1) A(2) A(3)
- B(0) B(1) B(2) B(3)
- C(0) C(1) C(2) C(3)
- Join
Functional versus data partitioning

**Data partitioning:**
- Allows a high degree of parallelization for loops with high iteration count.
- Allows good distribution of workload across (homogeneous) processors.
- Loop-carried data dependencies can severely impact performance.

**Functional partitioning:**
- Best for separation of workload across heterogeneous processors.
- Inter-function data dependencies typically converted into buffered streams.

Choice is directed by data dependency patterns.
Example functional partitioning

```c
int A[N][M];

while (..)
{
    produce_img();
    consume_img();
}
```

```c
produce_img()
{
    for (i ...)
        for (j ...)
            A[i][j] = ...
}
```

```c
consume_img()
{
    for (i ...)
        for (j ...)
            ... = A[i][j];
}
```

Thread1: while (..)
    produce_img();

Thread2: while (..)
    consume_img();

Synchronize thread progress:
- **True dependency**: consumer must wait for valid data
- **Anti dependency**: producer must wait with over-writing until after consumption
Function pipelining: synchronization

```c
int A[N][M];

while (..)
{
    produce_img();
    consume_img();
}

produce_img()
{
    for (i ...)
    {
        for (j ...)
            A[i][j] = ...
    }
}

consume_img()
{
    for (i ...)
    {
        for (j ...)
            ... = A[i][j];
    }
}

Channel ch;

Thread1: while (..)
        produce_img();
        consume_img();

Thread2: while (..)
        produce_img();
        consume_img()
{
    for (i ...)
    {
        for (j ...)
            write_int(ch, ...)
    }
}

consumer_img()
{
    for (i ...)
    {
        for (j ...)
            ... = read_int(ch);
    }
}
```

Channel access functions implement thread stall.
Pipeline dependency analysis

Potential pipelining showed in colors, with resulting Fifo's
Function pipelining: Channel APIs

Too many choices for channel-based communication:
- Standard Java util.concurrent queue classes
- Intel’s TBB (C++) queues
- Linux 'pipes' and 'sockets'
- OpenCL channels
- OpenMAX IL for streaming media processing
- MPI message-passing channels
- . . .

Very different queue implementations:
- Inter-thread, inside process memory context
- Inter-process, inside shared-memory system
- Inter-system, through device interfaces

NOTE: C++ STL queues are NOT thread-safe!
Example data partitioning

```c
int sum = 0;
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Distribute the workload over multiple cores.
- Each core handles part of the loop index space.

```c
int sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++) {
    int value = some_work(i);
    sum += value;
}
```

- Workload scales nicely across multiple cores
- Easy to write down 😊, but hard to grasp all consequences!
- Highly dangerous, might cause extremely hard-to-track bugs! 😞
Application Analysis
Finding data dependencies

Vector Fabrics’ approach:

- Compile program source code, compiler is adapted for instrumentation.
- Execute the instrumented program:
  - Traps all memory load/store operations: match ld/st operations that address the same memory location
  - Relates ld/st operations with nested loop structure: separate loop-carried dependencies from loop in-bound and loop out-bound dependencies
  - Builds an execution profile (call tree), across file boundaries
- Analyze loops with their data dependencies for parallelization patterns
Recognize parallelization patterns

Analyze loops with data dependencies for parallelization patterns:
- Reduction expressions
- Induction expressions
- Streaming dependencies, allowing data duplication and localization

Avoid considering ‘false’ memory dependencies:
- Local variables on stack, duplicated through thread local storage
- Re-use of memory locations through `malloc()` and `free()`.

Relate data dependencies and patterns to locations in C(++) source code for required code transformations.
Example: LAMMPS molecular simulator

- Source code configured for sequential version.
- About 187K lines of C++ source code in 636 files.
Example: LAMMPS data dependencies

- Parallelization opportunity detected in loop over particles, inside loop over time steps
- Different dependency patterns shown in different colors
Parallel performance prediction

- Estimate overhead from thread fork/join and synchronization
- Estimate execution schedule with loop-carried dependencies

Speedup of this loop: 3.9x, overall speedup: 2.4x
Today’s gap: multi-processor machines are everywhere, yet multi-threaded programming is difficult and error-prone.

Proper tooling is required to avoid (data race-) errors and obtain insight in performance issues. Obtain such insight before spending time on re-coding for parallelization.

Various tools are available today. They do support real-world application analysis.
**Today's gap**: multi-core CPUs and multi-processor machines are everywhere, yet multi-threaded programming is difficult and error-prone.

Proper tooling is required to avoid (data race-) errors and obtain insight in performance issues. Obtain such insight before spending time on re-coding for parallelization.

Various tools are available today. They do support real-world application analysis.
Thank you

Check [www.vectorfabrics.com](http://www.vectorfabrics.com) for a free trial of concurrency analysis