Parallel Programming and Optimization with GCC

Diego Novillo
dnovillo@google.com

Outline

- Parallelism models
- Architectural overview
- Parallelism features in GCC
- Optimizing large programs
  - Whole program mode
  - Profile guided optimizations

Parallel Computing

Use hardware concurrency for increased

– Performance

– Problem size

Two main models

– Shared memory

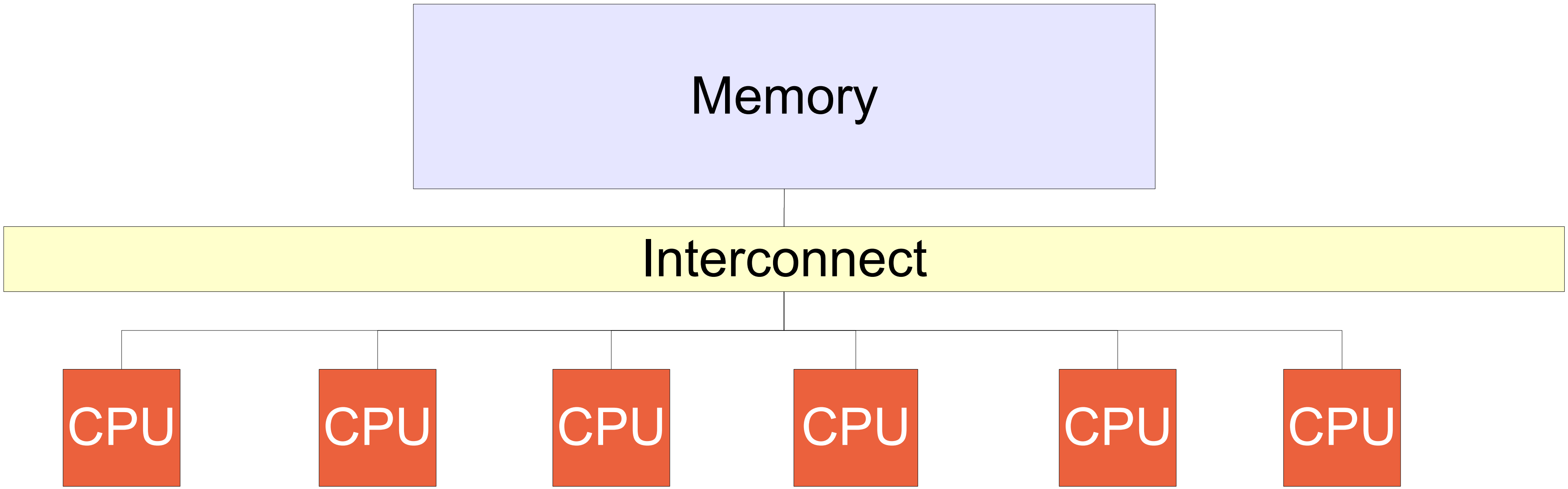
– Distributed memory

Nature of problem dictates

– Computation/communication ratio

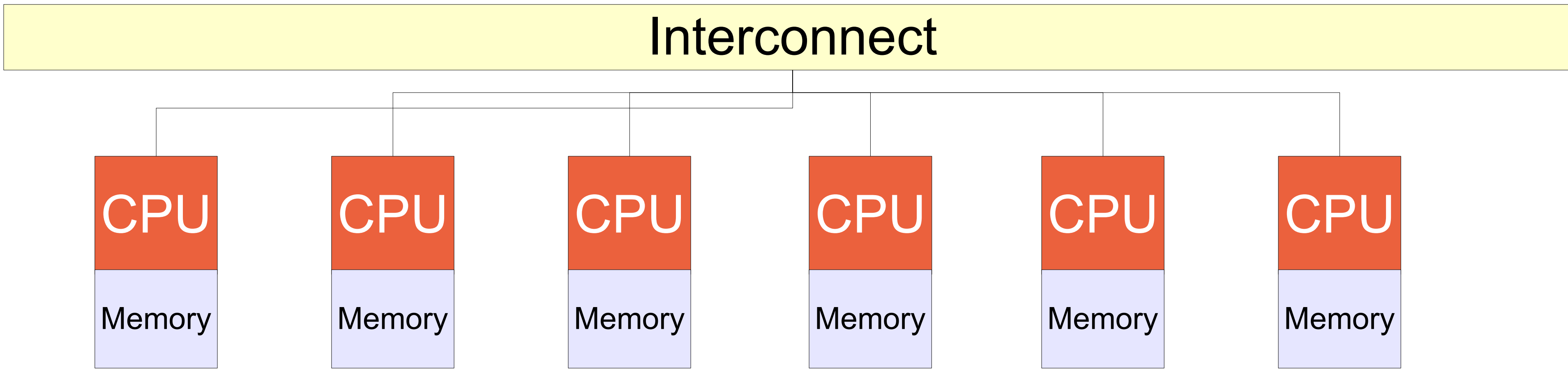
– Hardware requirements

Shared Memory



- Processors share common memory
- Implicit communication
- Explicit synchronization
- Simple to program but hidden side-effects

Distributed Memory

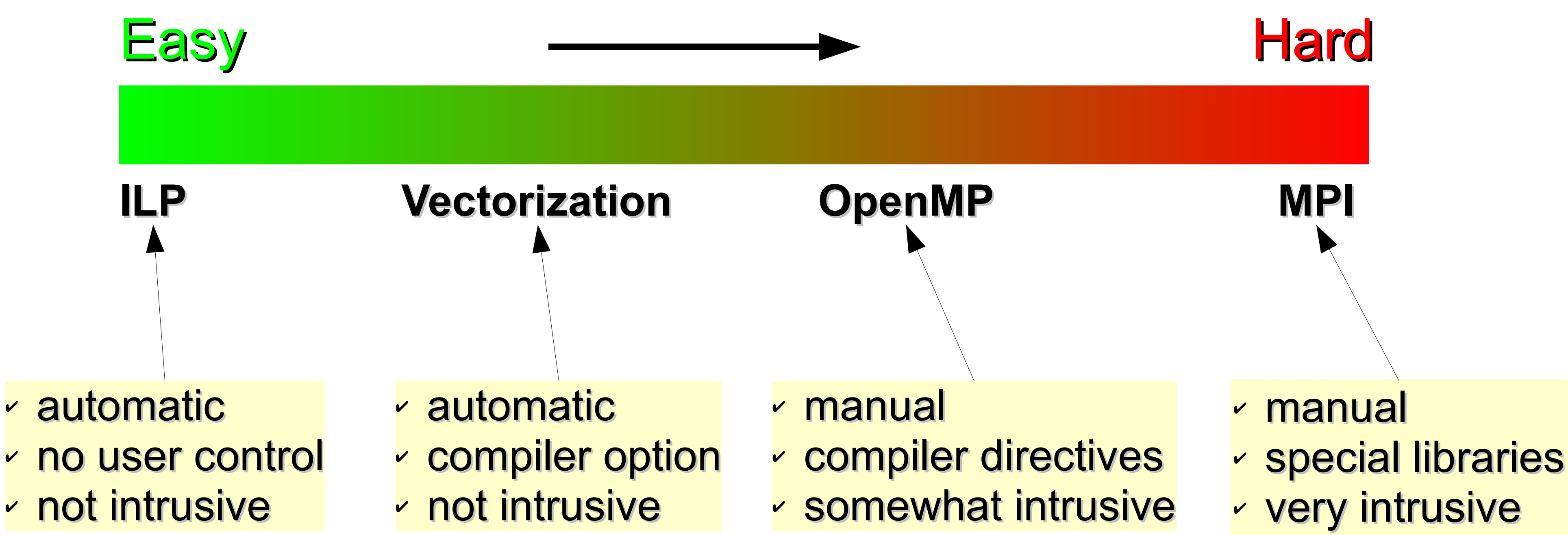


- Each processor has its own private memory
- Explicit communication
- Explicit synchronization
- Difficult to program but no/few hidden side-effects

Parallelism in GCC

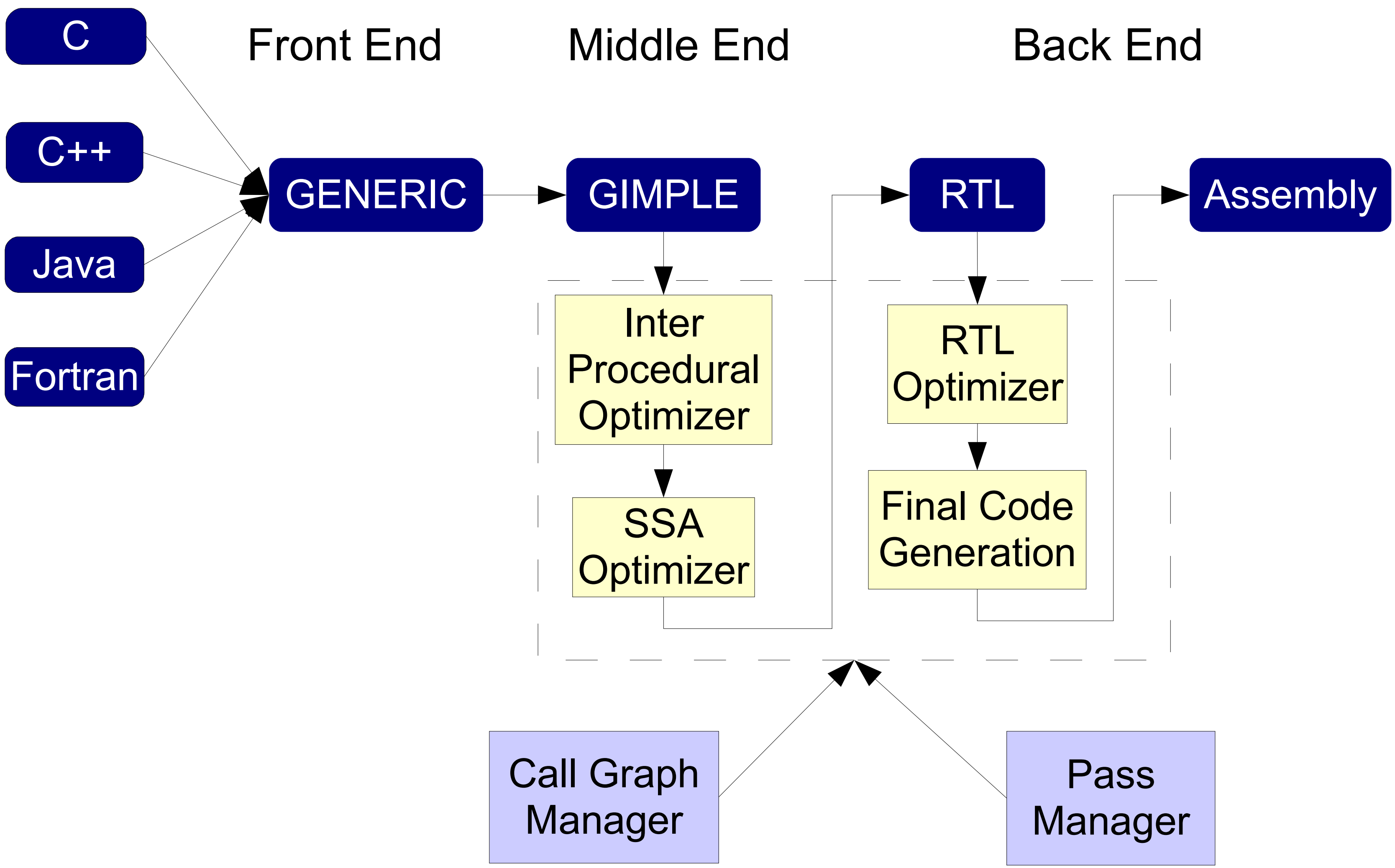
Google

GCC supports four concurrency models



Ease of use not necessarily related to speedups!

GCC Architecture



Vectorization

Perform multiple array computations at once

Two distinct phases

- Analysis → high-level
- Transformation → low-level

Successful analysis depends on

- Data dependency analysis
- Alias analysis
- Pattern matching

Suitable only on loop intensive code

Vectorization

(~2x on P4)

Vectorized

```
.L2:
        movdqa   c(%eax), %xmm0
        paddd    b(%eax), %xmm0
        movdqa   %xmm0, a(%eax)
        addl     $16, %eax
        cmpl     $1024, %eax
        jne      .L2
```

```
int a[256], b[256], c[256];

foo ()
{
  for (i = 0; i < 256; i++)
   a[i] = b[i] + c[i];
}
```

Scalar

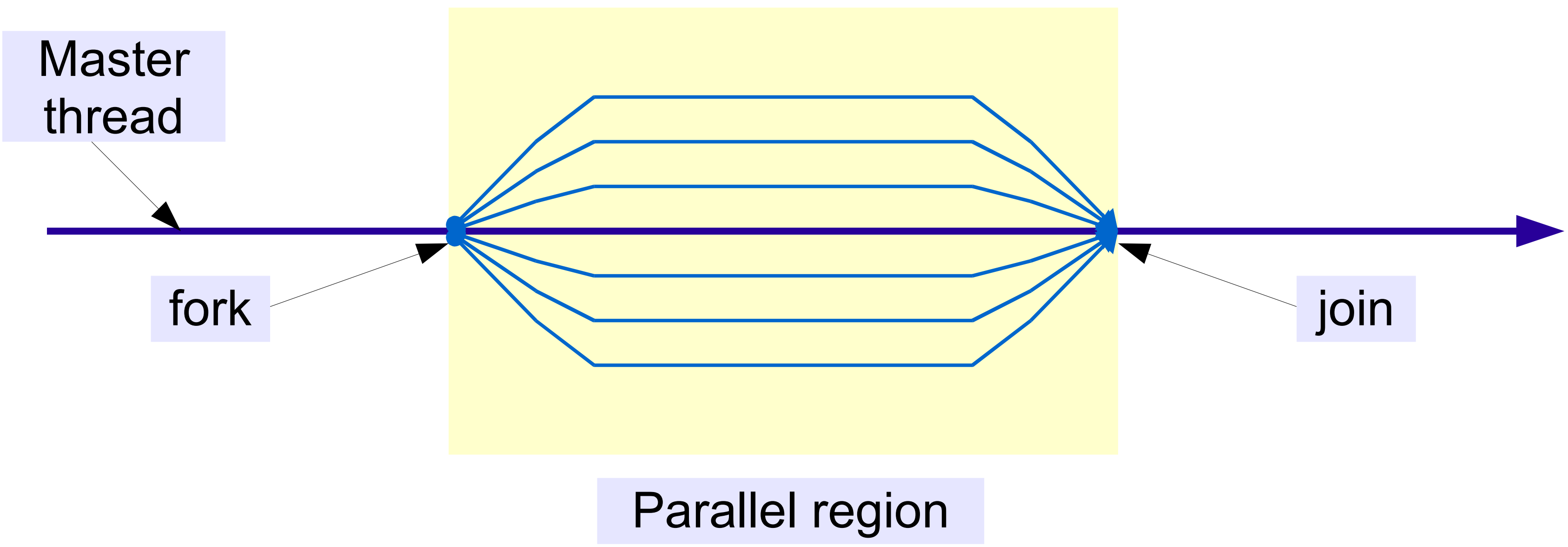
```
.L2:
        movl     c(,%edx,4), %eax
        addl     b(,%edx,4), %eax
        movl     %eax, a(,%edx,4)
        addl     $1, %edx
        cmpl     $256, %edx
        jne      .L2
```

OpenMP – Programming Model

Based on fork/join semantics

- Master thread spawns teams of children threads
- All threads share common memory

Allows sequential and parallel execution



OpenMP - Hello World

```
#include <omp.h>
main()
{
  #pragma omp parallel
  printf ("[%d] Hello\n", omp_get_thread_num());
}
```

```
$ gcc -fopenmp -o hello hello.c
$ ./hello
[2] Hello
[3] Hello
[0] Hello   ← Master thread
[1] Hello
```

```
$ gcc -o hello hello.c
$ ./hello
[0] Hello
```

Optimization Options

Level	Transformations	Speed	Debuggability
-O0	None (default)	Slow	Very good
-O1	Few	Not so fast	Good
-O2	Many	Fast	Poor
-Os	Same as -O2 + size	N/A	Poor
-O3	Most	Faster	Very poor
-O4	Nothing beyond -O3	N/A	N/A

It may be faster than -O2 due to smaller footprint

Optimization Options

- Optimizations done at two levels
  - Target independent, controlled with -f
  - Target dependent, controlled with -m
- There are more than 100 passes
- Not all can be controlled with -f/-m
- -Ox is **not** equivalent to a bunch of -f/-m
- Use -fverbose-asm -save-temps to determine what flags were enabled by -Ox
- Use -fno-... to disable a specific pass

Enabling additional optimizations

Not every available optimization is enabled by -Ox

```
-ftree-vectorize
-ftree-loop-linear
-ftree-loop-im
-funswitch-loops (-O3)
-funroll-loops
-finline-functions (-O3)
-ffast-math
```

Hundreds of -f and -m flags in the documentation

Optimizing Very Large Programs

f1.C

```
foo()
{
   for (;;) {
     ...
     x += g (f (i, j), f (j, i));
     ...
   }
}
```

f2.C

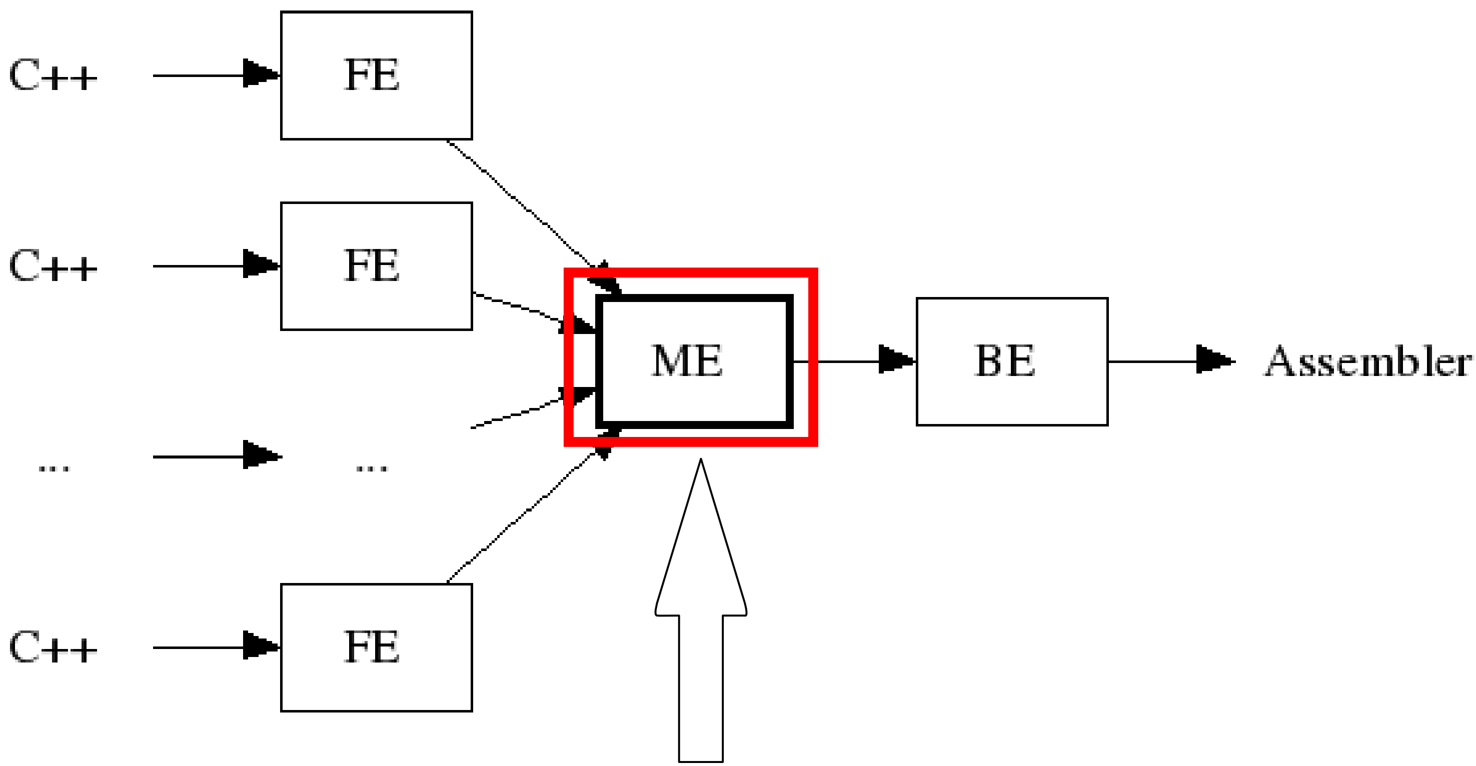
```
float f(float i, float j)
{
  return i * (i - j);
}

float g(float x, float y)
{
  return x - y;
}
```

Optimizations are limited by the amount of code that the compiler can see at once

Current technology only works across one file at a time

Compiler must be able to work across file boundaries

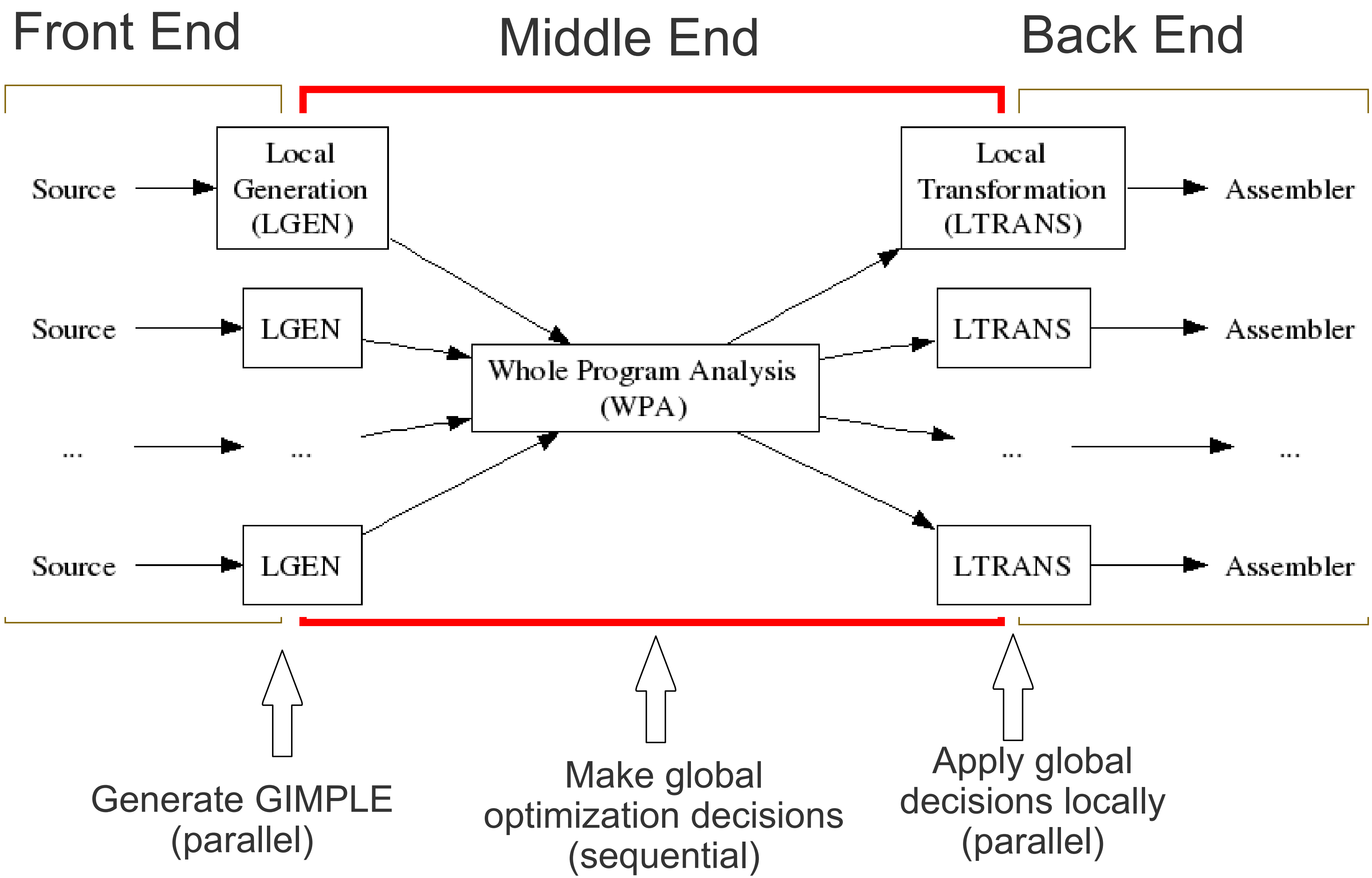
Optimizing Very Large Programs



Problem

Thousands of files, millions of functions, tens of gigabytes
Massive memory/computation complexity for a single machine

WHOPR Architecture - 1



WHOPR Architecture - 2

Compilation proceeds in 3 main phases:

- LGEN (Local GENeration)
  - Writes out GIMPLE
  - Produces summary information

- WPA (Whole Program Analysis)
  - Reads summary information
  - Aggregates local callgraphs into global callgraph
  - Produces global optimization plan

- LTRANS (Local TRANSformation)
  - Applies global optimization plan to individual files
  - Performs intra-procedural optimizations
  - Generates final code

WHOPR Architecture - 3

- Phases 1 (LGEN) and 3 (LTRANS) are massively parallel

- Phase 2 (WPA) is fan-in/fan-out serialization point
  - Only operates with call graph and symbols
  - Transitive closure analysis not computationally expensive

- Scalability provided by splitting analysis and final code generation
  - Restricts types of applicable optimizations
  - For smaller applications, LTRANS provides full IPA functionality (whole program in memory)

Profile Guided Optimization

- Three phases
  - Profile code generation: Compile with -fprofile-generate
  - Training run: Run code as usual
  - Feedback optimization: Recompile with -fprofile-use
- Allows very aggressive optimizations based on accurate cost models
  - Provided that training run is representative!
- Compilation process significantly more expensive
- May not be applicable in all cases

GProf

Probes inserted automatically by compiler

Compile and link application with -pg

Run application as usual

Use gprof to analyze output file gmon.out

```
$ gcc -pg -O2 -o matmul matmul.c
$ ./matmul
$ gprof ./matmul
```

OProfile

- System-wide profiler.
- No modifications to source code
- Samples hardware counters to collect profiling information
- User specifies which hardware counter to sample
- Needs super-user access to start
- Start Oprofiler daemon
- Run application
- Use reporting program to read collected profile

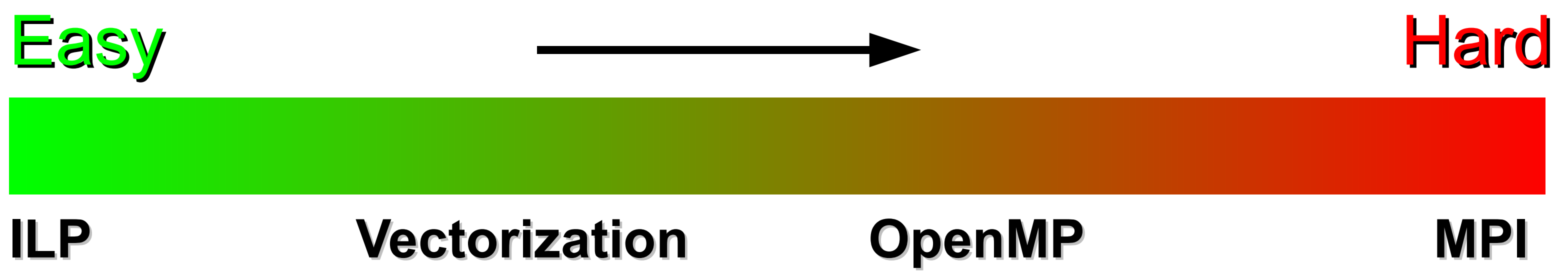
Profile Guided Optimization Advances

- Instrument → Run → Recompile cycle too demanding
- New feature being developed to use hardware counters

1. Program compiled as usual
2. Runs in production environment with hardware counters enabled
3. Subsequent recompilations use profile information from hardware counters

This allows for always-on, transparent profile feedback

Conclusions



- There is no “right” choice
- Granularity of work main indicator
- Evaluate complexity ↔ speedup trade-offs
- Combined approach for complex applications
- Algorithms matter!
- Good sequential algorithms may make bad parallel ones

Conclusions

- Performance tuning goes beyond random compiler flags
- Profiling tools are important to study behaviour
- Each tool is best suited for a specific usage
  - Try different flags and use /usr/bin/time to measure
  - Oprofile → system wide
  - Gprof → intrusive but useful to isolate profiling scope
  - Compiler dumps to determine source of problem
- New advances in instrumentation and whole program compilation will simplify things