Comma C/C++
A parallel programming language

By

Steve Casselman

CEO and Founder

Comma Corp.

Parallel Computing Technology

Computers and compilers evolved together

First there was the hardware. The hardware was programmed by hand, entering opcodes and data by flipping switches and swapping patch cords. It was the good old days.

Then someone came up with the card reader and assembly language and life was good.

In the 1970's powerful mini computers were developed. The need for a higher level language spurred the creation of a sequential language, C, that matched the sequential hardware architecture.

Microprocessors have been matched to these sequential languages ever since.

The language evolved to match the hardware

C and C++ are the most popular programming languages. Billions of lines of code have been written in C/C++. In a world that has gone parallel, C/C++ are sequential programming languages.

Many people think that a new language is needed. One that is designed to be parallel from the ground up. But nobody wants to rewrite billions and billions of lines of code.

If C/C++ were a naturally parallel language everyone would be happy.

The Problem: Sequential languages match sequential hardware.

The C and C++ languages are the most successful languages because they match the behavior of the single core microcomputer. The CPUs and compilers evolved together and were very efficient for a long time. Compilers got better at mapping algorithms to a single core and cores got faster. Life was good until the system broke. Cores could go no faster and many core machines were the only way to build bigger and more powerful computers

What the world needs now: A parallel language that looks exactly like C/C++

When C was first developed it contained an operator that implemented a list semantic that allowed the programmer to write a list of expressions. This operator was defined in a way that made it essentially useless. It is still part of the C/C++ language today.

The Comma!

Solution:
What the world needs
Comma C

We will interpret comma (or list) operator “,” as the “parallelism operator” The semi-colon “;” will be the synchronization token. Billions of lines of code can be reused with the only changes being the replacement of some semi-colons with commas.

The details:
Comma C looks exactly like Regular C

1) A=B;
   C=D;
   E=F;

2) A=B,C=D, E=F;

Both are forms of standard ANSI C

Syntax and semantics of the semi-colon and comma are the same.

The comma operator in C

```
int A = 0;
int B = 1;
int C = 2;
                        // evaluate sequentially
A=B + C, B=A+C; // right to left => A = 4 and B = 2
                        // left to right => A = 3 and B = 5
```

ANSI C says to evaluate left to right. This gives the comma have the same semantics as the semi-colon.

The usage of the comma meaningless and is almost never used in programs today. The comma works just like the semi-colon!

There is a third way to evaluate the above equations.

In Parallel

Keep the syntax but change the semantics

The Third option: Propagate variable's values to the compound statement and then evaluate all statements in parallel.

If A = 0 and B = 1 and C = 2

A=B+C, B=A+C;

implies

A=1+2, B=0+2;

Therefore:

A = 3 and B = 2

Remember before A = 4 and B = 2 or A = 3 and B = 5

This describes CommaC's fine grained parallelism.

Write Pipelined Code in Comma C

```
void
f(int a, int *b)
{
*b = a;
}
void
g(int b, int *c)
{
        *c = b;
}
void
h(int c, int *d) {
        *d = c;
}
main() {
int a = 0, b = 1, c = 2, d = 3;
printf("a =%d, b=%d, c=%d,\
          d=%d\n",a,b,c,d);
for (a=0,a<10,a++)
   f(a,&b), g(b,&c), h(c,&d),
   printf("a =%d, b=%d, c=%d,\
          d=%d\n",a,b,c,d);
}
```

Output:

a=0, b=1, c=2, d=3
a=1, b=0, c=1, d=2
a=2, b=1, c=0, d=1
a=3, b=2, c=1, d=0
a=4, b=3, c=2, d=1
a=5, b=4, c=3, d=2
a=6, b=5, c=4, d=3
a=7, b=6, c=5, d=4
a=8, b=7, c=6, d=5
a=9, b=8, c=7, d=6

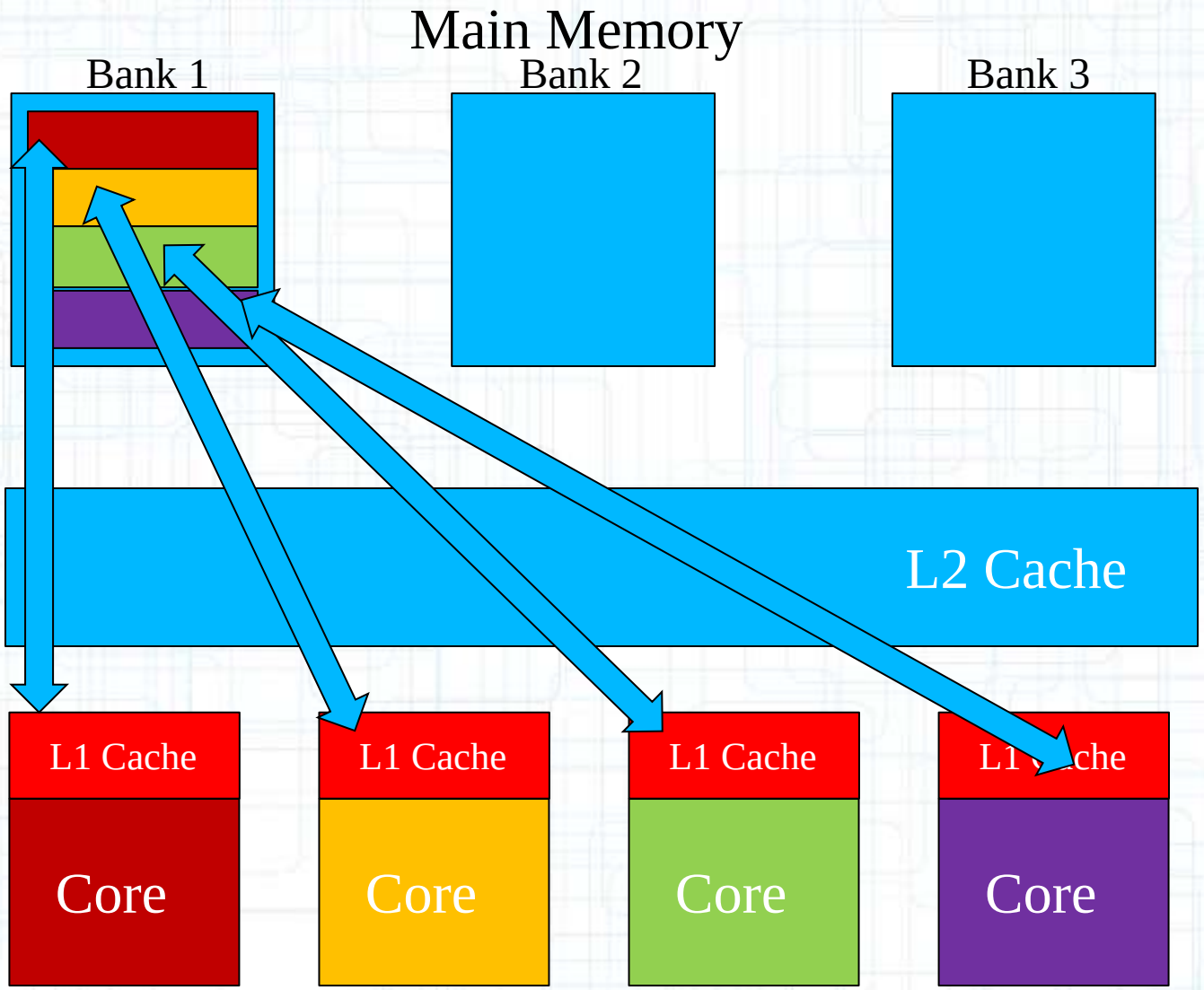
Pipelining Data Flow

Data flow in modern languages favors data parallel and task parallel kinds of activities.

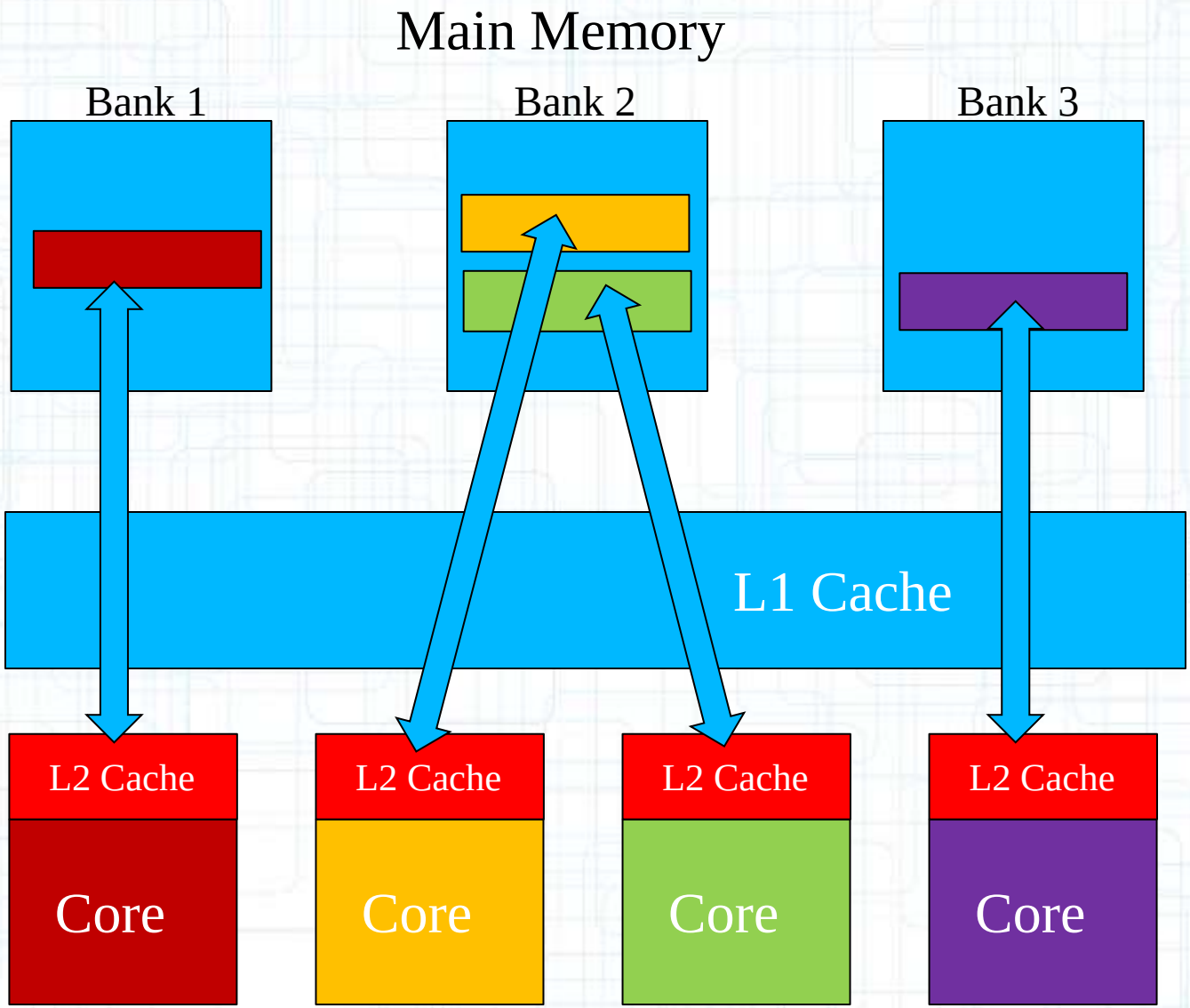
It is possible to write programs that encourage pipelined data movement but it's very hard.

It would be better if the language treated pipelined data movement as a "first class citizen". Pipeline data movement saves power by not moving the data as far as data and task parallel data movement.

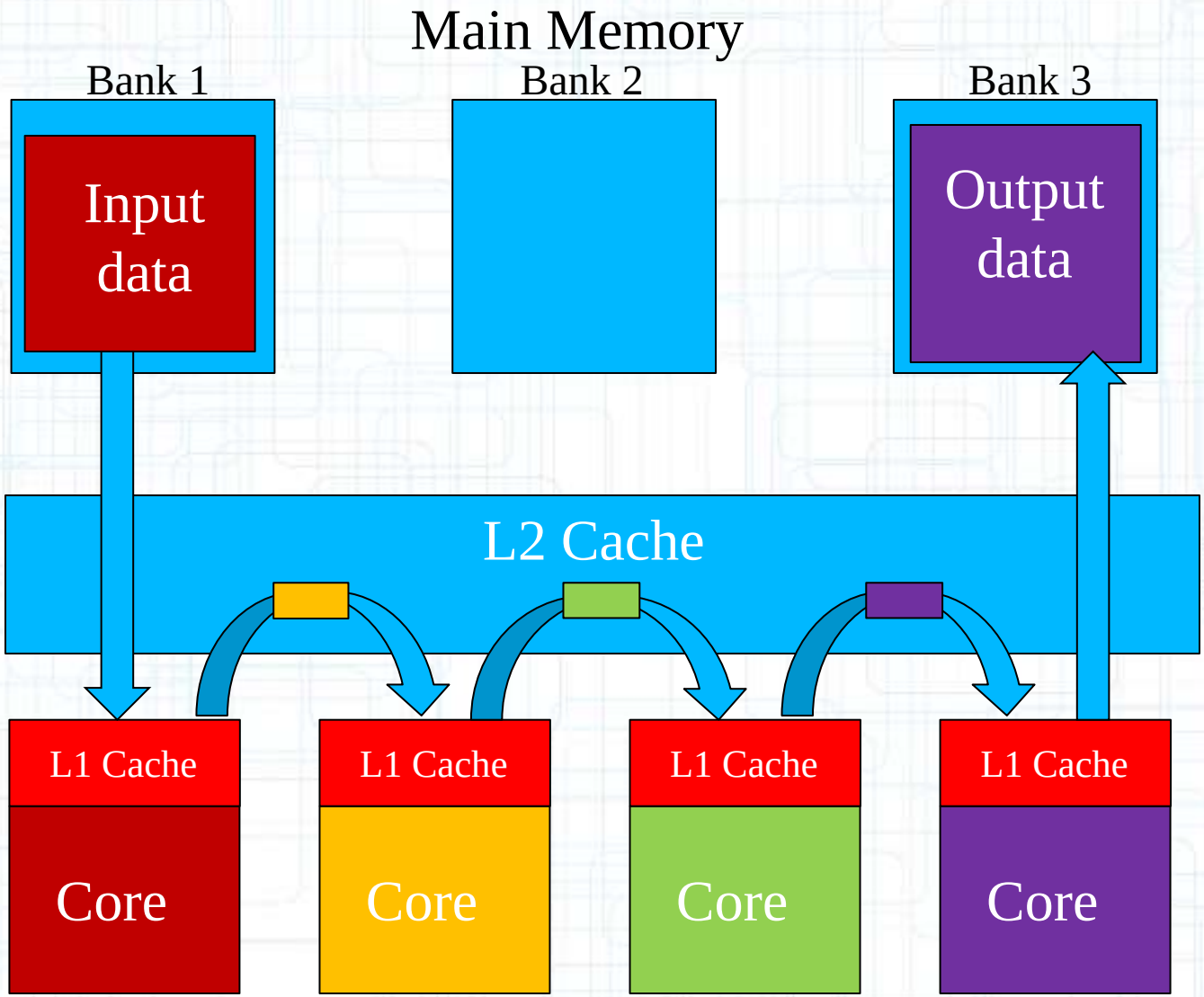
Data Parallel: Data shared among cores. Lots of data collisions, easy to program.



Task Parallel: Tasks have their own memories too many tasks thrash the memory. Harder to program



Pipeline Parallel: Data flows one way, no data collisions in main memory. Very hard to program.



Pipeline Parallelism improves performance

Results are reused.

In most runtime strategies data is computed on and then returned to memory.

By reusing the results of a calculation you save on data movement and thus a savings in time and energy. This means higher performance *and* lower power.

A language that has pipeline parallelism as part of it's syntax allows the programmer to express that parallelism more deeply and effectively.

Comma C Language Features:

Efficient expression of fine grain and pipelined coding styles.

Efficient memory model.

Easy transition for C programmers.

C programs that don't use the comma are Comma C compatible.

Works with other languages.

Lower power algorithm execution.

Conclusions

Comma C is an easy and natural way to describe pipelined parallelism, as well as fine grained parallelism, in a C/C++ looking and feel program.

Programmers already know Comma C. It is just as easy to teach Comma C as it is to teach ANSI C.

Comma C lets the programmer think in parallel. It lets the programmer express parallelism easily and lets the compiler do a better job at speeding up a program.