Compile and Runtime Padding: An Approach to Realising Synchronous MIMD Execution

IFIP’94
Hamburg 30 aug 1994

Lasse Natvig
Dept. of Computer Systems and Telematics
Norwegian Institute of Technology
University of Trondheim
Norway
Presentation overview

- Background
- Synchronous MIMD programming
- Padding
  - compile time
  - parametrised
  - runtime
  - programmer helped
- Related work
- Relevance / discussion of padding
- A standard PRAM / BSP programming environment

E-mail: Lasse.Natvig@idt.unit.no
Background

● The gap between theory and practice in (parallel) computing
● Are parallel algorithms from TCS efficient in practice?
● What is hidden behind the Big-O notation?
● Asymptotical analysis assumes infinite problems
● Comparing algorithms for finite problem sizes
  ➞ requires PRAM programming
  ➞ notation by James Wyllie (79)
  ➞ PRAM (Fortune and Wyllie) is synchronous MIMD
Cole’s parallel merge sort vs. Batcher’s bitonic sort

- Implemented in a prototype high-level synchronous MIMD language on a simple PRAM simulator
  - spin-off: experience in synchronous MIMD programming

- Result:
  - Batcher’s $O(\log^2 n)$ algorithm outperforms Cole’s $O(\log n)$ parallel merge sort when $n$ is smaller than $10^{20}$
  - Not a proof, but indication of performance in practice
  - Reported in
    - Proceedings of SUPERCOMPUTING’90, New York, November 1990.
    - Dr.Ing.-thesis (Ref. [3])
Synchronous MIMD programming

- Outlined by James Wyllie in 1979
- Synchronous statement:
  "If each processor in a set of processors P begins to execute a statement S at the same time, all processors in P will complete their execution of S simultaneously"

- Advantages
  - Flexibility and efficiency of MIMD
  - Full control over the processors, *i.e.* easy programming

- Problems
  - Difficult to realise efficiently for a large number of processors
    - synchronous processors
    - shared global memory
Padding --- overview

- Straight compile time padding
- Parametrised compile time padding
- Runtime padding
- Programmer-helped padding
- The last way out: Barrier synchronisation
Straight Compile Time Padding

(1) IF Cond(ProcLocalVar) THEN
(2)   DoBigJob
(3) ELSE
(4) BEGIN
(5)   DoSmallJob;
(6)   Wait(TC(DoBigJob) - TC(DoSmallJob));
(7) END

● Assumptions
  ■ the time consumption of every instruction in # clock ticks is known
  ■ synchronous processors (clock ticks)
(1) PROCEDURE A(Limit : INTEGER);
(2) VAR Count : INTEGER;
(3) BEGIN
(4)  FOR Count := 1 TO Limit DO
(5)    IF Cond(ProcLocalVar) THEN
(6)       DoBigJob { Assumed CTD }
(7)    ELSE
(8)       BEGIN
(9)         DoSmallJob; {Assumed CTD}
(10)      Wait(TC(DoBigJob) - TC(DoSmallJob));
(11)    END
(12) END.

● CTD = compile time deterministic, i.e. the time consumption can be calculated at compile time
CONST Lower = 4;
SHAREVAR UpperLimit : INTEGER;
PRIVATEVAR Local : INTEGER;
PROCEDURE B(a,b:INTEGER;
x,y:INTEGER);
VAR i:INTEGER;
BEGIN
  IF Condition(Local) THEN BEGIN
    FOR i := a TO b DO
      LoopJob; {Assumed CTD}
    Wait(x);
  END
  ELSE BEGIN
    ElseJob; {Assumed CTD}
    Wait(y);
  END
END;{of B}
BEGIN {Main program}
  {Code executed in parallel}
  B(Lower + 10, 20, x1,y1);
  B(50, UpperLimit);
  B(80, Local);
  BarrierSynchronisation;
  ...
END
(1) PRIVATE VAR s, k, m : INTEGER;
(2) PRIVATE VAR ProcId : ProcNo;
(3) SHARED VAR x : INTEGER;
(4) ...
(5) PROCEDURE C(a, b : INTEGER);
(6) VAR i : INTEGER;
(7) BEGIN
(8)   IF b < 100 THEN
(9)     FOR i := 1 TO a DO
(10)    b := b * 10;
(11)   ELSE
(12)     Wait(x);
(13) END;{of C}
...

(14) BEGIN { Main program }
(15)   k := ProcLocalComputation;
(16)   s := AnySharedIntegerVariable;
(17)   IF ProcId = 0 THEN
(18)     X := s * TC(line (10));
(19) ELSE
(20)     Wait(TC(line (18)));
(21)   FOR m := 1 TO 100 DO
(22)     C(s, k);
(23) END;
Programmer-helped padding and general synchronisation

- **Automatic insertion of barrier synchronisation**
  - invisible in early phases of program development
  - visible through compiler option
    - trained programmers might want to rewrite the code (*i.e.* tuning)
      - help the compiler
      - manual explicit padding
  - a profiler should give performance information about the extra code (padding or barrier) to help the compiler and/or the programmer to choose the right alternative
Padding is efficient on a simulator!

- When padding can be used, it gives synchronous behaviour with no or very little \(O(1)\) additional execution time, while a barrier synchronisation takes \(O(\log n)\) additional time on a CREW PRAM.

- The only (?) problem is the assumption of synchronous processors at the instruction level.

- BUT, synchronous processors are easily implemented on a simulator:
  - development, evaluation, teaching and sharing of algorithms
  - early phase of the development of a complex parallel system.
Can padding be used to give efficient execution of synchronous MIMD programs?

- SHLS with padding
- SHLS with barrier-sync (bs)

PRAM-assembler w/NOP’s

PRAM-assembler w/bs-calls

BSP-machine simulator

CRAY-C90
PARAGON
BSP-hw

SHLS = Synchronous High-Level Statement
Related work

- University of Saarbrucken
  - SB-PRAM computer
  - FORK language

- Oxford University
  - BSP Library, GL

- University of Karlsruhe
  - Modula-2* & 3*, Triton

- University of Joenssu, Finland
  - pm2, w / some padding
  - PRAM emulator

- University of Trondheim
  - Synchronous MIMD programming
  - PRAM pascal w/ some padding
    - Bjørn Tore Dale
    - Ove K. Pettersen
  - At present, no activity ...

E-mail: Lasse.Natvig@idt.unit.no
A standard PRAM/BSP programming environment

- GL
- PRAM-pascal
- PRAM-C++
- PRAM-model assembler standard
- “Bulkyfying”
- Machine-X
- PRAM-simulator standard
- PC or WS with DOS or UNIX
- Textbook: parallel algorithms

E-mail: Lasse.Natvig@idt.unit.no