Data Parallel Programming I

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

SIMD Computers

- SIMD = Single Instruction, Multiple Data
  - Array processors,
  - Processor arrays,
  - "Data parallel systems".
- Architecture
  - 1 control unit (CU) attached to host computer,
  - n processors = arithmetic units (AU) + local memories,
  - Interconnection network.

"Parallel vector" = set of variables duplicated on each processor.

Operation Principle

- CU broadcasts instruction to all processors.

  Single program executed by processor array.

- Each ALU executes instruction on local data
  - arithmetic/logic,
  - read/store data from/on host,
  - transfer data through network.

- CU may select by bitmask *subset* of processor array for execution.

  Conditional execution.

Same instruction executed on all processors in lock-step.

MasPar MP-1

University of Vienna

- DEC Unix workstation as host
- 1024 = 32*32 processors ($\approx$ 16384)
  - 40 * 32 bit registers,
  - 4 bit integer ALU,
  - 64 bit FPU,
  - 16 KB of RAM,
  - 15 MHz clock.
- X-NET mesh interconnection

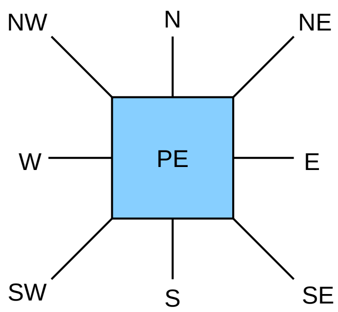
X-Net

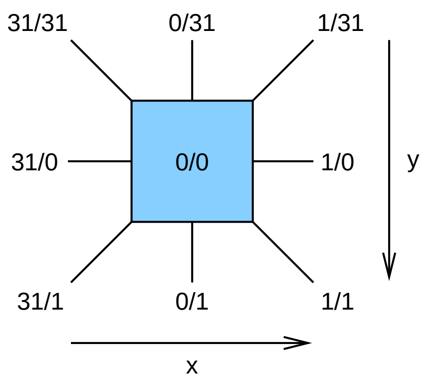
- 32 * 32 processors

  proc = ⟨ixproc/iyproc⟩
  0 ≈ ixproc < nxproc
  0 ≈ iyproc < nyproc

- 2D grid (torus)

  wrap-around connections at boundaries





MasPar Programming

- Parallelizing Fortran Compiler

  $\approx$ vectorizing compiler.

- MPL Maspar Programming Language

  "Data Parallel C"
  Based on GNU C compiler

```
PATH=$PATH:/usr/maspar/bin
mpl -O test.m -o test
mpconfig
mpstat
mpl_cc
```

MPL Declarations

- Singular declarations

  ```
  int i, a[N];
  ```

  variable allocated on ACU/host <u>once</u>

- Plural declarations

  ```
  plural int i, a[N];
  ```

  variable allocated on <u>each</u> PE

- Predefined constants
  - Singular: `nxproc`, `nyproc`

    Size of processor array.
  - Plural: `ixproc`, `iyproc`

    Coordinates of current PE.

MPL Statements

- Singular and plural statements
  - A *plural statement* references at least one plural variable.
  - A *singular statement* does not reference any plural variable.
- Notion: "Active Processor Set"
  - All *singular* statements are only executed on ACU.
  - All *plural* statements are executed on all PEs of the active processor set.

Assignments

```
int i, j, k[nxproc][nyproc];
plural int a, b;
```

- `i = i+j;` singular
- `a = a*i+b;` plural
  - ACU broadcasts i to all PEs.
  - Each active PE executes statement using its local variables a and b.
- `j = a*i+b;` illegal!

  All PEs would store their results in singular j!
- `k[ixproc][iyproc] = a*i+b;` plural

  Each PE stores its result in a different location of the singular matrix on the ACU.

X-Net Communication

Plural by definition.

Expression `xnet␣[d].var`

- *var* . . . a plural variable
- d . . . distance from current PE
- ␣ . . . direction

  N, NE, E, SE, S, SW, W, NW.

Every reference to an xnet variable results in a data transfer from/to the denoted processor.

X-Net Communication

```
plural int a, b;
```

- `xnetW[1].a = a;` left shift

  All active processors simultaneously transfer the values of their local variables a to their left neighbors.

- `a = xnetE[1].a;` left shift

  Each active processor receives the new value for its local variable a from the local variable a of its right neighbor.

- `a = a+xnetE[1].b;`

  Each active processor increments its local variable a by the value of the local variable b of its right neighbor.

All communications take place simultaneously; at boundaries of processor array, connections are "wrapped around"!

Example

(see additional slide)

- Store value in DPU,
- Shift elements one PE south,
- Store result in host array.

Side Effects

- I/O and system-calls only on ACU!
- What's wrong with the following?

```
plural a;
a = random();
```

Conditionals

```
if (plural-exp)
    ...
else
    ...
```

1. Save value of mask register (MR).
2. Evaluate plural expression on each active PE.
3. Set MR to result of evaluation

   Active processor set = set of processors where MR is set.
4. Execute `then` branch on active processor set.
5. Revert MR.
6. Execute `else` branch on active processor set.
7. Restore saved value of MR.

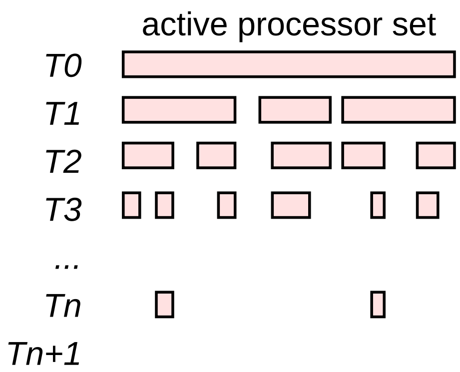
Both branches are evaluated sequentially!
T(if) = T(exp)+T(then)+T(else)

Loops

```
while (plural-exp) ...
```

1. Save value of mask register (MR).
2. Evaluate plural expression on each active PE and set MR to evaluation result.
3. While active processor set is not empty

   Execute body on active processor set;
   Evaluate plural expression on each active PE and set MR to evaluation result.
4. Restore saved value of MR.



All PEs are blocked until the last PE has completed the while loop!

MasPar Program

Data-parallel programming.

- Structure like sequential program.
- Plural declarations.
- Plural statements executed in parallel.
- Active processor set constrained by conditional and loop constructs.

Only a single flow of control!