High Performance Computing: Architecture, Applications, and SE Issues

Peter Strazdins

Department of Computer Science,

Australian National University

e-mail: peter@cs.anu.edu.au

May 17, 2004

1 Overview of High Performance Computing

- many applications have huge computational, memory and I/O requirements
- given a data size n, typically the execution time is $t(n) = O(n^2)$
  - challenge of HPC is in large n, not in complexity of $t(n)$
  - often (eg. vector operations) $t(n) = a_0 + a_1 n$; the values of a_0, a_1 are important!
- recent dramatic improvements in HPC technology has made these possible
  - will there be a limit to the demands?
- HPC machines are:
  1. capable of 'fast' computing rates
  2. fickle: small code transformations can make huge differences in performance!
  3. range from state-of-the-art PC's to supercomputers

2 Why High Performance Computing?

- who needs to understand HPC?
  - programmers, seeking to adapt (tune) their code to a new machine
  - systems designers, for computing systems where performance is important
  - managers, needing to choose the best machine for their division
  - computational scientists and engineers
- requires an overall understanding of computer performance:
  - how algorithms interact with the architecture
  - underlying principle: PARALLELISM on many scales
- HPC machines at ANU:
  - 2–12 processor Sun UltraSPARC shared memory parallel processors (eg. `iwaki`)
  - various (Linux) Beowulf cluster computers, e.g. the 196-CPU Bunyip
  - the AlphaServer SC, 127×4 distributed / shared memory parallel processor

3 HPC Architectures: RISC (Scalar) Microprocessors

- low (entry) costs (eg. PCs); dramatic performance increases over last decade
- features:
  - pipelined instruction execution:
    - break instr'n execution into k stages; $\Rightarrow$ can get $\leq k$-way ||ism
    - issues: branch instr'ns & dependencies between instr'ns
  - multiple instruction issue:
    - a small group of (different) instr'ns are scheduled to *execute together*
    - eg. UltraSPARC: $\left\{ \begin{array}{l} \leq 2 \text{ floating point} \\ \leq 1 \text{ load / store} \ ; \ \leq 1 \text{ branch} \\ \leq 2 \text{ integer} \end{array} \right\}$ (max. 4) instr'ns / group
  - a deep memory hierarchy using cache memory to 'hide' long memory access times:
    - idea: get data that is *"currently most needed"* is brought into a (smaller) faster storage area
    - requires programs be (re-)written to re-use data (and on each level)

4 HPC Architectures: Vector Processors

- vector arithmetic can encompass most compute-intensive 'scientific' applications:
  - requires innermost loops to be of the form: $\vec{V} := f(a, b, c, \ldots, \vec{V}, \vec{X}, \vec{Y}, \ldots)$
  - eg. daxpy loop $\vec{Y} \leftarrow \vec{Y} + a\vec{X}$:

```
DO i = 1, N
  y(i) = y(i) + a * x(i)
ENDDO
```

  can be expressed in Fortran 90 as `y(1:N) = y(1:N) + a * x(1:N)`

- vector processors have large sets of vector registers (eg. 32×256 words) & vector instructions (have high startup costs (a_0) $\Rightarrow$ need large n for good performance!)
- vector operations are regular: can be deeply pipelined and overlapped
  - different (eg. 8) 'segments' can also be executed in || !
- use multiple banks of fast (expensive) memory for || access (no caches)
- high entry cost, but mature technology; once synonymous with supercomputing
  - the Earth Simulator (still the world's fastest) has 640×8 vector CPUs

5 HPC Architectures: Parallel Processors

- idea: to get more speed, connect p (scalar or vector) processors (cells) together, to co-operate to achieve a single task
- Shared Memory Parallel Processors
  - a small ($p \leq 64$) number of cells, with separate local caches and a shared global memory
  - need cache coherency (so that all cells have a consistent view of the memory) and synchronization (to keep all cells co-ordinated)
  - SMP (Symmetric MultiProcessors) are now widely supported by vendors
- Distributed Memory Parallel Processors
  - a large (typically $10^2 \leq p \leq 10^3$) number of cells, connected by a communication network
  - cells communicate (and synchronize) via messages
  - programming is more difficult:
    - must be in terms of each cell's local address space
    - need to use explicit message passing calls

6 HPC Applications

- many applications require enormous computational resources:
  - Quantum Chromodynamics
  - gene sequencing & protein folding
  - Materials Science / Structural Engineering
  - Speech/Vision Processing
  - weather forecasting / climate modelling

    3-D 1 km^3 grid, 6 variables, 1 min. time steps:
    $\approx 10^{10}$ cells and 100 FLOPs / cell $\Rightarrow$ 8 GFLOPS rate required !
  - chess (Deep Blue)
  - multimedia applications, Virtual Environments
    - require high memory performance; must also achieve a minimum 'Quality of Service'
  - database and web servers
    - need high memory & I/O performance; serve each transaction as a separate processes

7 Programming Paradigms HPC Machines

- illustrate by Game of Life:

```
for (;;) {                                          // iterate forever
  for (i=0; i<N; i++) for (j=0; j<N; j++)           // iterate over each element of grid
    t[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1] == 2);
  for (i=0; i<N; i++) for (j=0; j<N; j++)
    A[i][j] = t[i][j];
}
```

- data parallel model (for all architectural types), eg. Fortran 90 (`n = N+2`)

```
WHILE .TRUE. DO                          // repeat forever
  T(2:n-1,2:n-1) = A(1:n-2,2:n-1) + A(2:n-1,1:n-2) +
                   A(3:n,  2:n-1) + A(2:n-1,3:n)
  WHERE (T.eq.2) A = 1                   // apply over all elements of array
  ELSEWHERE A = 0
END WHILE
```

- for distributed memory parallel processors, need to specify how data is *placed*
  - in order to get an even workload, and to minimize communication
  - eg. in High Performance Fortran, do this when declaring matrix:

```
DIMENSION A(N, N), T(N, N)                  // declare space for matrices
!$HPF$ PROCESSORS P(2,2)                   // arrange 4 cells in a 2 x 2 grid
!$HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO P: A
```

- programming parallel processors more directly is not nice:
  - in (pseudo) 'shared memory C':

```
int A[N][N], t[N][N];          // shared data
local int i, j, id=0;          // each cells gets new copy
...
id = p_fork(4);                // creates a '|| region' of 4 cell; 0 <= id < 4
                               // id is each cells identifier
for (;;) {
  for (i=0; i<N; i++) for (j=0; j<N; j++)
    if (Owns(id, i, j, N))     // ie. this cell 'owns' element A[i][j]
      t[i][j] = (A[i][j-1] + A[i-1][j] +
                 A[i][j+1] + A[i+1][j]) == 2;
  barrier_sync();              // wait till all other cells get here
  for (i=0; i<N; i++) for (j=0; j<N; j++)
    if (Owns(id, i, j, N))
      A[i][j] = t[i][j];
  barrier_sync();
}
```

  - in message passing C (assumes 1 data element per cell):

```
int a;                                          // this cell's element of A
for (;;) {
  int an, as, ae, aw;
  n_send(NORTH, a);    n_send(SOUTH, a);        // send 1 word to neighbour cells
  n_send(EAST, a);     n_send(WEST, a);
  n_recv(NORTH, an);   n_recv(SOUTH, as);       // now receive from neighbours
  n_recv(EAST, ae);    n_recv(WEST, aw);
  a = (an + as + ae + aw) == 2;
}
```

8 Software Engineering Issues

- for optimum performance, program must be optimized for each feature of the machine
  - with the exception of vector processors, compiler technology is generally inadequate to do (all of) this (without help)
  - as the architectures continue to become more complex, this puts more & more burden on the software developers
- applications are complex and operate on huge data sets
  - parallel processors exacerbate the testing and debugging problems enormously!

  however, reliability is still crucial...
- legacy codes have been developed by a broad spectrum of people in a broad spectrum of programming paradigms for a broad spectrum of HPC machines...
  - portability of software is still a major issue
- a long-term solution:
  - specify the overall computation of the data objects; allow compiler / run-time system / hardware to optimize

9 Trends and Conclusions

- exponential increase in microprocessor performance (Moore's Law) expected to continue over the next decade
  - HPC still will be a relatively volatile field
  - but cost of developing a new chip also increasing exponentially . . .
- while many supercomputing companies have closed, vendors are widely supporting Symmetric Multiprocessors and clusters of workstations models of parallel processors
  - seeking the commercial market; multimedia applications of increasing interest
  - the long development time typically required is still a problem . . .
- outstanding software engineering challenges:
  - to *properly apply* SE to the development of HPC applications
  - to provide better tools/environments (parallelizing compilers & runtime systems, debuggers performance analysers)
  - to solve the problem of portability (performance, as well as correctness), across not only current but future HPC machines