

## Parallel Computing and Data Locality

Gary Howell

In 1986, I had degrees in math and engineering and found I wanted to compute things. What I've mostly found is that:

Real estate and efficient computation have something in common.

### --location, location, location

At that time, the main problem in using supercomputers efficiently was keeping the CPU supplied. Cray vector computers had high peak flop rates, but few (almost no) codes could attain it.

The problem has spread to serial machines. Much (most) of the effort in getting efficient codes is in adapting them to keep the CPU supplied.

hand out Computer Magazine article -- but in the meantime, codes are inefficient because the CPU is not busy.

To get a "message" from RAM on a P4 requires approximately (assuming 60 nanosecond access to RAM, 2 flops per clock cycle, and a 2.8 GHz clock, 133 MHz PCI 32 bit wide PCI bus)

$$T_a = 300 + (\text{single precision number fetched}) * 40$$

Where  $T_a$  is the number of floating point adds or multiplies that could have been done during the data fetch.

By fetching a bunch of numbers next to each other, we can avoid the "300" latency. And then if we can reuse the number for 40 floating point adds or multiplies before we send it back to RAM, then (possibly) we can use the processor at full speed.

Serial performance--

- rule 1 don't swap out of memory - reading from a H/D takes "forever"
- rule 2 don't swap out of L2 cache (bring data into L2 cache in blocks)
- rule 3 don't swap out of L1 cache
- rule 4 don't swap out of registers.

How? Reuse whatever data is fetched for computations.

One way to get cache efficiency is to use the BLAS libraries. BLAS 1 operations are vector-vector such as

$$x \leftarrow x + \beta y \quad \text{or} \quad \langle x, y \rangle$$

BLAS-1 calls were originally helpful in using vector machines. For a vector vector operation, two vectors are fetched and typically for two vectors of length  $n$ ,  $2n$  flops are performed, i.e., one flop per fetch.

For a BLAS-2 operation (e.g., a matrix vector operation `_gemv`)

$$y \leftarrow y + Ax$$

a matrix and a vector (or two) are fetched. For a `_gemv`, 2 flops are performed for each element of the matrix, i.e. two flops per fetch.

For a BLAS-3 operation (e.g., a matrix- matrix multiply `_gemm`)

$$C \leftarrow C + AB$$

blocking allows data in cache to be reused many times. So matrix matrix multiplies can work at full clock speed. In particular, LU and QR factorization can be written in terms of `_gemm`, so work at full clock speed.

FFTs can also get pretty close to clock speed.

Anything else?

Typically, when I teach parallel computing I spend the first few weeks getting students to try to make BLAS-2 matrix vector multiplies go fast.

exercises

how big is A to swap out of RAM to the hard drive. What happens to the flop rate?

can you find the size of L2 cache by varying the size of A?

What difference does it make if you access data row wise when it's stored by columns?

loop unrolling, stride, compiler flags, inlining

compare speed of BLAS 1, BLAS 2, BLAS 3

Typically, the students get their BLAS-2 codes sped up from 1 or 2 % of peak speed to 10 or 20%. And then you introduce the tuned BLAS package to them and they find they can run a bit faster.

If you do matrix operations, consider downloading the ATLAS BLAS for your architecture.

The BLAS underlie the blocked algorithms implemented in LAPACK -- serial linear algebra (solution of  $Ax=b$ , least squares problems, eigenvalues, singular value decomposition)

Of course, there are other fundamental math libraries that speed basic computations, and which tend to be faster and more robust than codes you would write yourself. For example, UMFPACK, SuperLU, SPARSKIT for solving sparse systems of linear equations, or FFTPACK for FFTs.

Before we leave serial computation, how do you tune a serial code? You can use various timers to see which parts of a code are time consuming. A partial list would be `etime` in Fortran, Unix call `timex`, the `MPI_Wtime` wall time function, the C `clock` function, etc.

Alternatively, or first, we can use profilers such as `prof`, `gprof`. In parallel we can use `Vampir`.

So tune by:

- 1) find out where most of the computations are done
- 2) concentrate there
- 3) is there an optimized routine you can link to?

### **What does all this have to do with parallel computation?**

If the problem would be better solved with more computations or with a finer mesh than will fit into RAM, then we have to go for a parallel solution.

For the serial case, accessing memory required a time equivalent to

$$T_a = 300 + (\text{single precision numbers fetched}) * 40$$

Flops. For a shared memory machine, we might instead have

$$T_a = 1200 + (\text{single precision numbers fetched}) * 40 \quad ??$$

i.e., just a bit higher latency. Also, the shared memory machine is likely to have a larger cache memory than a typical serial machine. Actually the latency tends to grow with the number of processors sharing the memory.

For large machines, or for distributed memory machines, latency is yet higher. We might have

$$T_a = 10K + (\text{single precision numbers fetched}) * 40$$

Flops for a fast interconnect and perhaps more typically

$$T_a = 100K + (\text{single precision numbers fetched}) * 40$$

to fetch data on RAM on another node. For comparison, the time to access data on the hard drive might correspond to

$$T_a = 10M + (\text{single precision numbers fetched}) * 200$$

flops. **So it's much cheaper to access data from the RAM of another CPU than to access data on a hard drive.**

For either a shared or distributed memory machine, an additional complication is the uncertainty of using a shared resource. The memory access times corresponding to the formulae  $T_a$  are usual, but sporadically they are large. This can be due to Unix daemons running in the background, or to network saturation.

Reiterating. As a sole user of a machine, one has repeatability in timing results. Across a shared network? On a shared file system?

Parallel computing has been a CS topic for at least 30 years. Only in the last ten years have we developed means for writing codes which survive the current flavor of machine. These are libraries which allow cooperation between processes. Some popular libraries for writing parallel programs are

**PVM** -- not a standard, but good for heterogeneous networks (inspiring current work in GRID computing)

Robust and simple. Can be used to convert a set of networked work stations to be a rudimentary cluster.

**MPI** -- Message Passing Interface -- the main standard and the main topic this week

**OpenMP** -- is a shared memory standard. "pragmas" are inserted into C or C++ or Fortran 77, 90, 95 converting serial programs to parallel. It is often possible to quickly parallelize with OpenMP, e.g., in an afternoon. (automatic parallelizers sometimes work well, they tend to throw in a few too many pragmas which the user removes). A typical success might run with a speedup of 4 on 8 processors and show rather little speedup beyond that.

Typically for many processors, shared memory really means NUMA, which are single system image but for which some memory is more quickly accessible depending on which group of processors the memory is attached to. Meaning that to scale to many processors, we still have to worry about data locality. The usual method is explicit message passing via MPI. One migration route is as follows:

**shmem** -- one sided or active communication-- Crays, SGIs, quadrics switches. simple in not having lots of commands often highly efficient where it exists, not a standard. If a code was originally on a Cray, it is likely to shmem.

**MPI-2** extends MPI to have some of the good features of PVM (spawning processes dynamically, linking to already running programs) and specifies implementations of one sided messages similar to shmem. It also provides some standard interfaces for parallel I/O.

All of the above have "bindings" to allow them to be called from either Fortran 77 90, or C C++.

Some other ways to implement parallel programs, HPF (high performance Fortran), co-array Fortran, UPC.

### **Shared (OpenMP) vs. Distributed Memory (MPI)**

It's often easy to get a parallelization for parallel machines. Many of the popular commercial scientific computation codes have been extended to work on shared memory machines. For example, the finite element code Abaqus has a shared memory version. It tends to give the best times with just a few processors. Shared memory allows solution of larger problems than would be possible with only the RAM from attached to a single CPU.

Distributed (MPI) are more versatile in the sense that one can always construct a parallel machine (use distributed workstations for example). Distributed machines are cheaper per processor. Massively parallel machines are distributed memory. MPI also run on shared memory machines. Large shared memory machines are NonUniform Memory Access (NUMA) so programming them to get a high degree of parallelism will typically mean MPI.

Hybrid OpenMP-MPI codes are possible and give a performance advantage on IBM SPs (but on SGLs, Crays, HP SuperDomdes, HP SC Alphas give no performance advantage, Beowulfs with 2 CPUs per mothernode? ).

### Writing Scalable Codes

If problem size is held constant and the number of processors is increased, then after a certain number of processors the code execution time starts to grow with the number of processors. Why?

Suppose that the serial execution time is

$$T = T_{\text{parallel}} + T_{\text{sequential}}$$

And that the parallel execution time for  $p$  processors is

$$T = T_{\text{parallel}}/p + T_{\text{sequential}} + T_{\text{communication}}(p)$$

Where time for communication increases with  $p$ . If  $T_{\text{parallel}}/T_{\text{sequential}}$  is 100 then the best possible speedup would be 100 (Amdahl's Law). But actually due to the time for communication, the best possible speedup would be less. And of course,  $T_{\text{parallel}}/T_{\text{sequential}}$  is often less than 100 and rarely greater.

**Because we use large parallel processors to solve large problems (or solve the same problem with a finer grid), it is often possible to efficiently use hundreds of processors.**

Suppose we keep the amount of data per processor constant as we increase the number of processors. If the computational time per processor stays constant as the number of processors  $p$  increases and if the communication costs grows only like  $\log(p)$ , then we say the job is scalable.

The efficiency of the parallel computation (for  $p$  processors) is then

$$E(p) = \frac{\text{(computational time on a processor)}}{\text{(computational time on a processor + const} \cdot \log(p) \text{)}}$$

If  $\text{const} \ll \text{computational time on a processor}$  then the efficiency will be nearly one.

Some tricks to getting an efficient parallel algorithm

- 1) have a lot of local data which requires few global updates
- 2) keep the amount of local data constant as the number of processors increases
- 3) minimize communication by lumping messages together
- 4) hide the latency of communication by prefetching data with nonblocking calls.
- 5) Use collective calls that perform communications in  $O(\log(p))$

If you want to see more about 3)-5) come to the MPI short course this week.

### Parallel Application Libraries

It's often saves time to use code others have developed (it's fun to reinvent the wheel

but not a good way to compete).

MPI was meant to spur development of parallel libraries. And it has.

DOE, NSF, and DOD have funded development of quite a few good parallel open source MPI based libraries.

SCALAPACK is parallel LAPACK (dense matrix algorithms)

SuperLU (direct solution of sparse matrix equations)

ARPACK and PARPACK (determination of a few eigenvalues and eigenvectors of a sparse matrix).

PETSc (extensible library for scientific computations)

pARMS (iterative solutions of sparse matrix equations by multilevel methods)

FFTW - (Fast Fourier Transforms)

Metis and Parmetis (grid partitioning algorithms)

This is not a complete list.

### **Parallel Programming Utilities**

Vampir is a parallel profiler. It helps us keep track not only of time spent in various subroutines, but also of time spent communicating. Works best for small numbers of processors. Knowing where time is spent helps in knowing where optimization is required.

TotalView is a parallel debugger. Allows us to step through parallel codes.

One good reference for MPI

Parallel Programming with MPI by Peter Pacheco, Morgan Kaufman Press.

Also you can find several good tutorials on the web. You can download one due To Rick Weed of ERDC.

```
ftp cs.fit.edu
login as anonymous
cd pub/howell
get
```