Calculating π in Parallel Using MPI

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Quantitative & Computational Biology
University of Southern California

Email: anakano@usc.edu

Objectives

1. **Task decomposition (parallel programming = who does what)**
2. **Scalability analysis: a key skill in parallel computing**

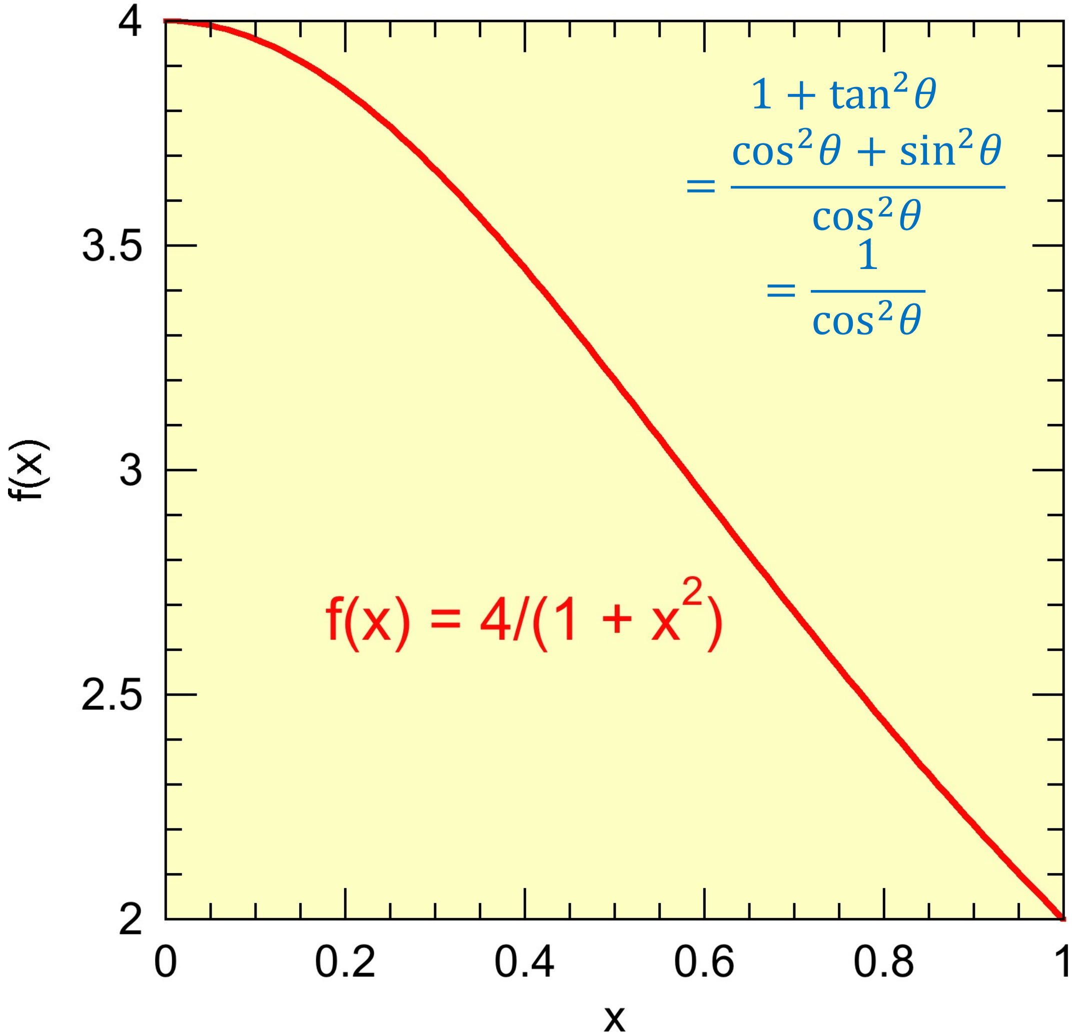
Integral Representation of π

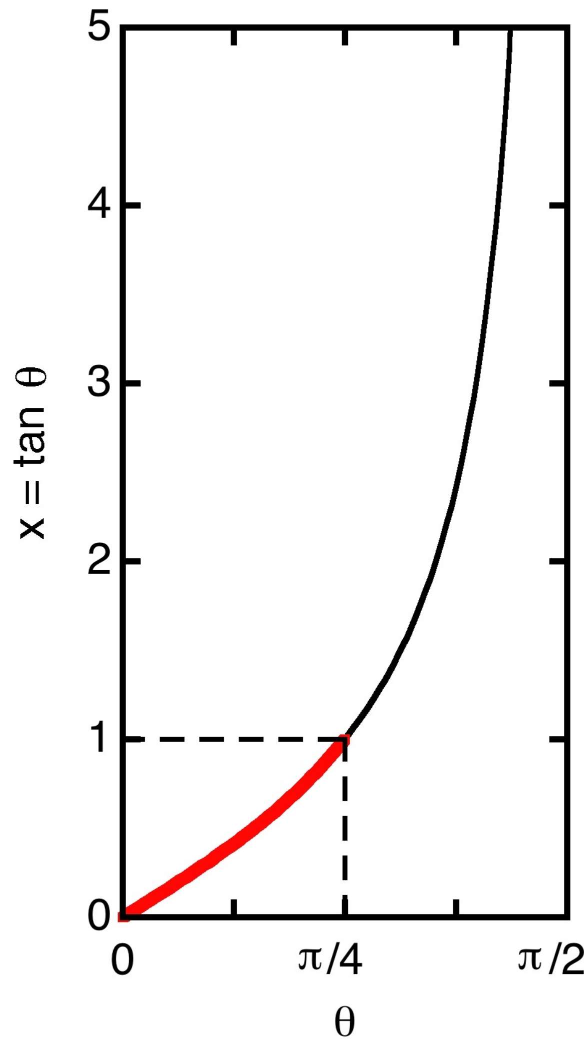
Coordinate transformation

$$\int_0^1 dx\,\frac{4}{1+x^2} = \int_0^{\pi/4} \frac{d\theta}{\cos^2\theta}\,\frac{4}{1+\tan^2\theta} = \int_0^{\pi/4} 4\,d\theta = \pi$$

$$x = \tan\theta$$

$$\frac{dx}{d\theta} = \frac{1}{\cos^2\theta}$$





Numerical Integration of π

- **Integration**

$$\int_0^1 dx \frac{4}{1+x^2} = \pi$$

- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta \; (i = 0,\ldots,N\text{-}1)$

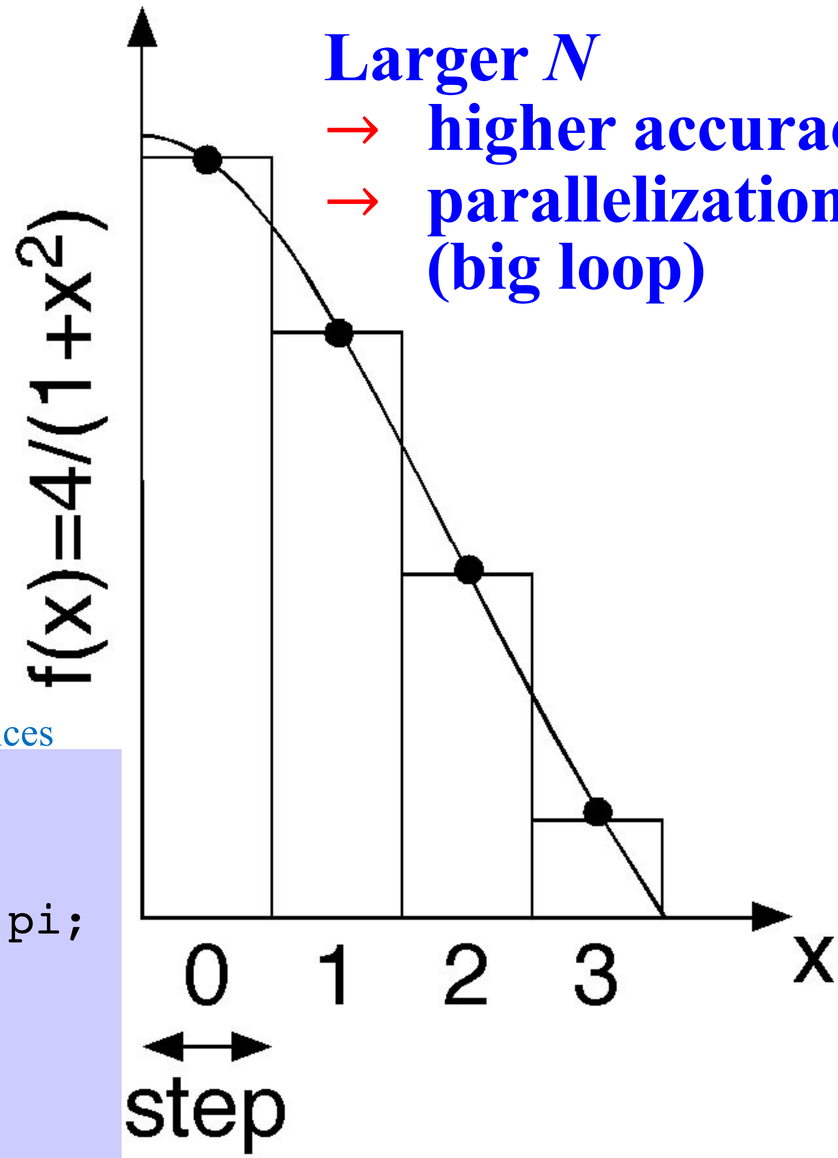
$$\sum_{i=0}^{N-1} \overbrace{\frac{4}{1+x_i^2}}^{\text{height}} \overbrace{\Delta}^{\text{base}} \cong \pi$$

Sum of rectangular areas

64-bit integer to handle large indices

```
#include <stdio.h>
#define NBIN 1000000000
void main() {
  long long i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);
  }
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```

A big loop to parallelize



Larger N
→ **higher accuracy**
→ **parallelization (big loop)**

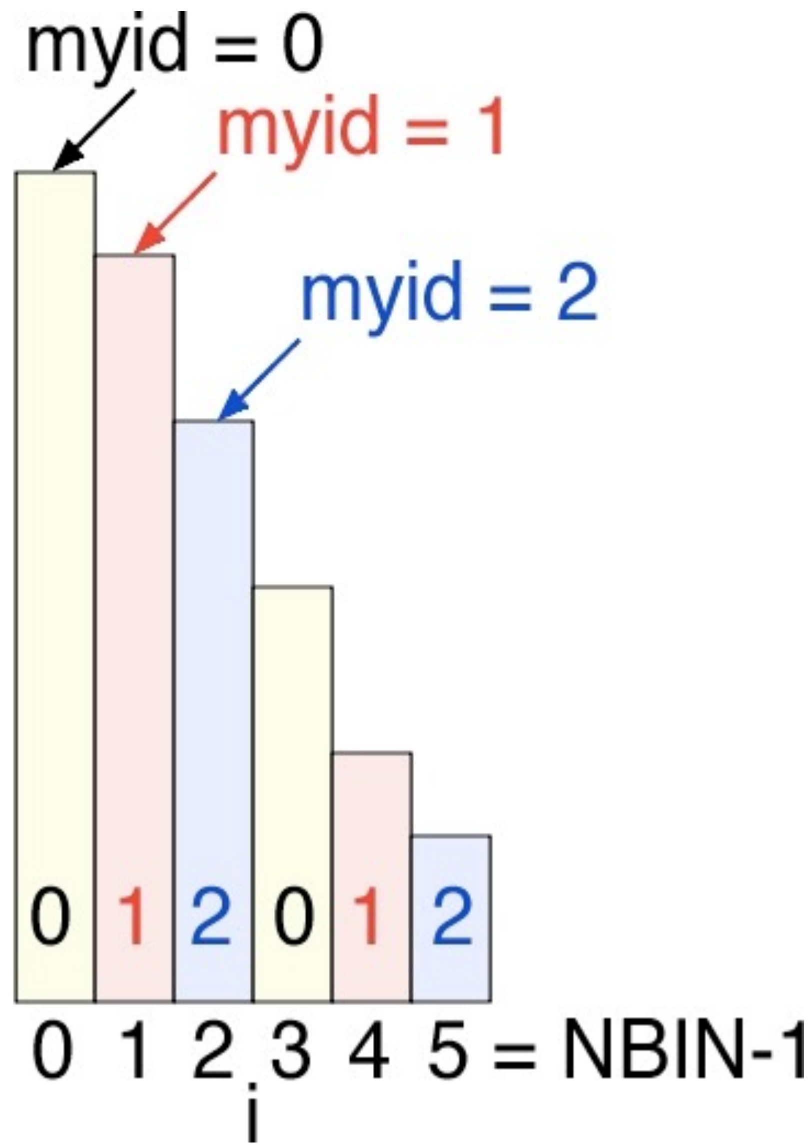
See https://en.wikipedia.org/wiki/Chronology_of_computation_of_%CF%80

Parallelization: Who Does What?

Interleaved assignment of quadrature points (bins) to MPI processes

Single program multiple data (SPMD)

```
...
for (i=myid; i<NBIN; i+=nprocs)
{
  x = (i+0.5)*step;
  sum += 4.0/(1.0+x*x);
}
partial = sum*step;
pi = global_sum(partial);
...
```



`MPI_Comm_rank(MPI_COMM_WORLD,&myid)`

myid = MPI rank

nprocs = Number of MPI processes

`MPI_Comm_size(MPI_COMM_WORLD,&nprocs)`

- Use `double MPI_Wtime()` to measure the running time in seconds

Measuring Runtime

`double MPI_Wtime():` Returns the elapsed wall-clock time in seconds since some time in the past. The "time in the past" is guaranteed not to change during the lifetime of the process

See p. 9 in https://aiichironakano.github.io/cs596/02MPI.pdf

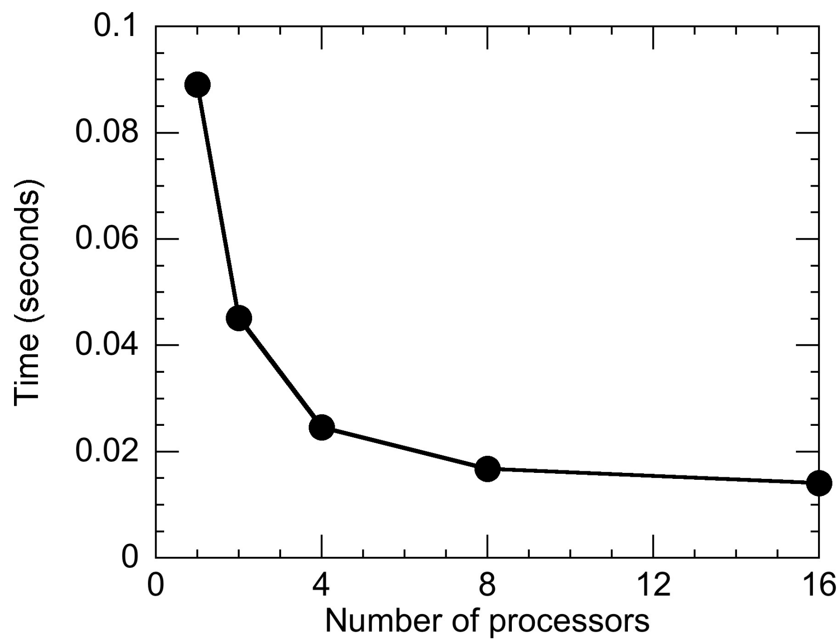
```
int main() {
  double cpu1, cpu2;

  MPI_Init(&argc, &argv);
  ...
  cpu1 = MPI_Wtime();
  /* Do computation here */
  cpu2 = MPI_Wtime();
  ...
  if (myid == 0) {
    printf("Nprocs & Global sum = %d %le\n", nprocs, pi);
    printf("Execution time (s) = %le\n",cpu2-cpu1);
  }
  ...
  MPI_Finalize();
  return 0;
}
```

Parallel Running Time

global_pi.c: NBIN = 10^7, on Discovery

```
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=1
...              16×1 = 16
mpirun -n $SLURM_NTASKS ./global_pi
mpirun -n             8 ./global_pi
mpirun -n             4 ./global_pi
mpirun -n             2 ./global_pi
mpirun -n             1 ./global_pi
```



Q: How Efficient Is the Parallel Program?

Scalability Analysis

- **Parallel computing = Solving a big problem (*W*) in a short time (*T*) using many processors (*P*)**

- **How *W*, *T* & *P* scale with each other?**
- **How to define the efficiency of a parallel program?**

See Grama *et al.*,
Chap. 5—Analytical modeling of parallel programs

Parallel Efficiency

- **Execution time:** $T(W,P)$
  W: **Workload**
  P: **Number of processors**

- **Speed:** $S(W,P) = \dfrac{W}{T(W,P)}$

- **Speedup:** $S_P = \dfrac{S(W_P,P)}{S(W_1,1)} = \dfrac{W_P T(W_1,1)}{W_1 T(W_P,P)}$

- **Efficiency:** $E_P = \dfrac{S_P}{P} = \dfrac{W_P T(W_1,1)}{P W_1 T(W_P,P)}$

Ideal speedup

How to scale W_P with P?

Fixed Problem-Size Scaling

$W_P = W$—constant (strong scaling)

- **Speedup:** $S_P = \dfrac{T(W,1)}{T(W,P)}$

$$S_P = \frac{S(W_P,P)}{S(W_1,1)} = \frac{W_P T(W_1,1)}{W_1 T(W_P,P)}$$

- **Efficiency:** $E_P = \dfrac{T(W,1)}{PT(W,P)}$

$$E_P = \frac{S_P}{P} = \frac{W_P T(W_1,1)}{P W_1 T(W_P,P)}$$

Solving the same problem faster using more processors!

- **Amdahl's law:** f (= sequential fraction of the workload) limits the asymptotic speedup

$$S_P = \frac{T(W,1)}{T(W,P)} \leq P$$

$$T(W,P) = fT(W,1) + \frac{(1-f)T(W,1)}{P}$$

$$\therefore S_P = \frac{T(W,1)}{T(W,P)} = \frac{1}{f + (1-f)/P}$$

$$\therefore S_P \to \frac{1}{f} \quad (P \to \infty)$$

Isogranular Scaling

$W_P = Pw$ **(weak scaling)**
w **= constant workload per processor (granularity)**

- **Speedup:** $S_P = \dfrac{S(P \bullet w, P)}{S(w,1)} = \dfrac{P \bullet w / T(P \bullet w, P)}{w / T(w,1)} = \dfrac{P \bullet T(w,1)}{T(P \bullet w, P)}$

- **Efficiency:** $E_P = \dfrac{S_P}{P} = \dfrac{T(w,1)}{T(P \bullet w, P)}$

$$S_P = \frac{S(W_P, P)}{S(W_1, 1)} = \frac{W_P T(W_1, 1)}{W_1 T(W_P, P)}$$

$$E_P = \frac{S_P}{P} = \frac{W_P T(W_1, 1)}{P W_1 T(W_P, P)}$$

Solving larger problems within the same time using more processors!

$$E_P = \frac{T(w,1)}{T(Pw,P)} \le 1$$

Analysis of Global_Pi Program

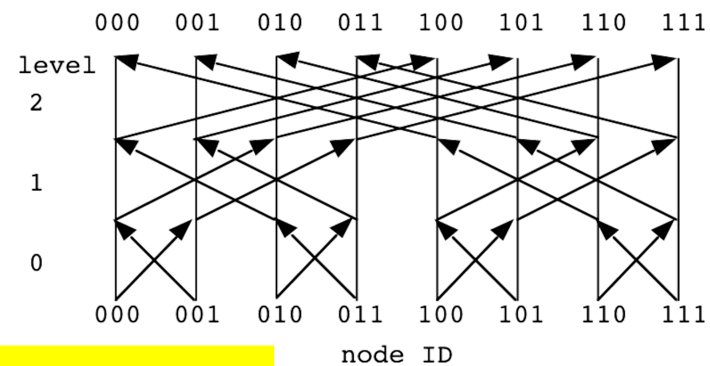
- **Workload ∝ Number of quadrature points, N (or `NBIN` in the program)**
- **Parallel execution time on P processors:**
  > **Local computation ∝ N/P**

```
for (i=myid; i<N; i+=P){
  x = (i+0.5)*step; partial += 4.0/(1.0+x*x);
}
```

  > **Butterfly computation/communication in global() ∝ log$P$**

```
for (l=0; l<log2P; ++l) {
  partner = myid XOR 2^l;
  send mydone to partner;
  receive hisdone from partner;
  mydone += hisdone
}
```

log$_2P$



P

$$T(N,P) = T_{\mathrm{comp}}(N,P) + T_{\mathrm{global}}(P)$$
$$= \alpha\frac{N}{P} + \beta\log P$$

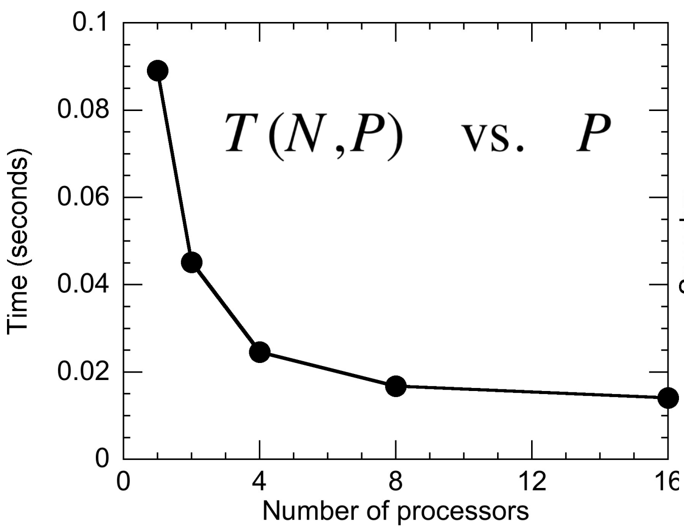
Fixed Problem-Size (Strong) Scaling

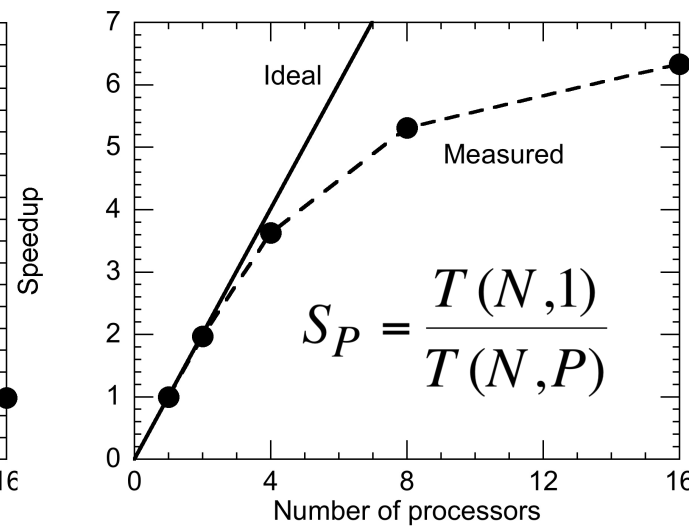
- **Speedup:**

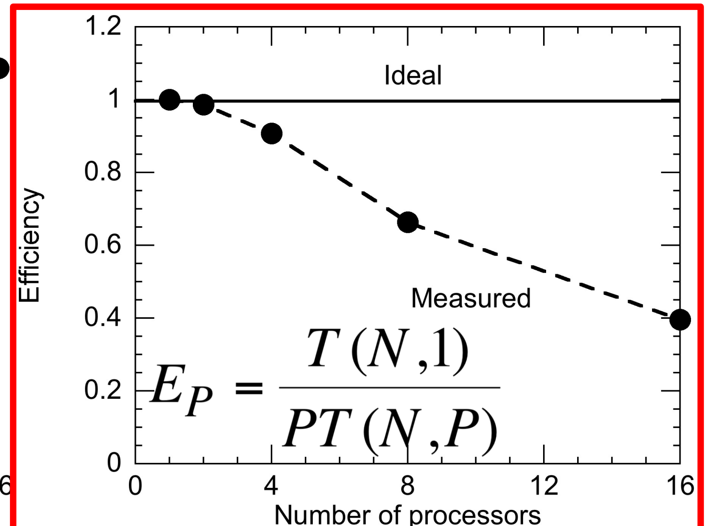
$$S_P = \frac{T(N,1)}{T(N,P)} = \frac{\alpha N}{\alpha \frac{N}{P} + \beta \log P} = \frac{P}{1 + \frac{\beta}{\alpha}\frac{P\log P}{N}}$$

- **Efficiency:**

$$E_P = \frac{S_P}{P} = \frac{1}{1 + \frac{\beta}{\alpha}\frac{P\log P}{N}}$$



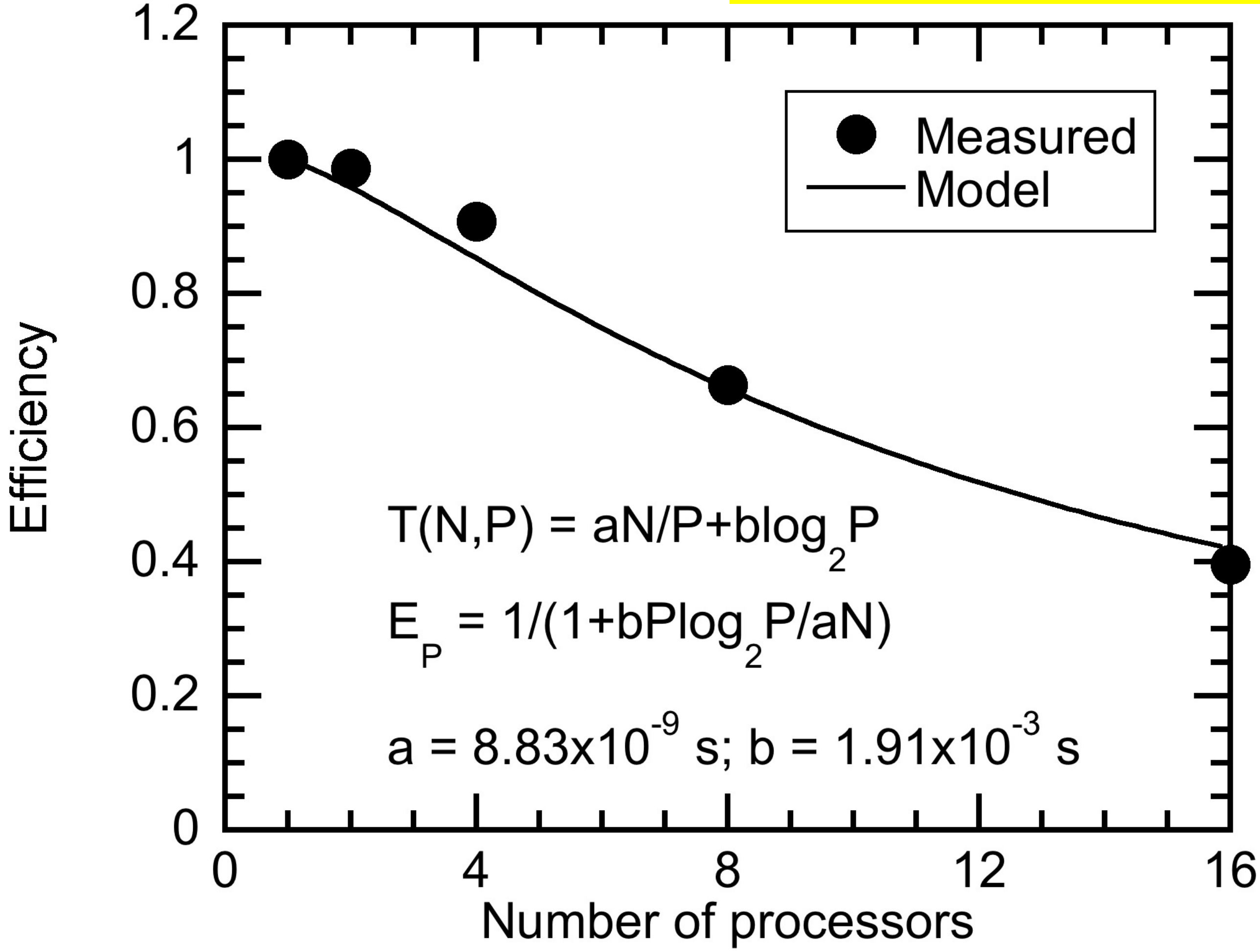




global_pi.c: $N = 10^7$, on Discovery

Fixed Problem-Size Scaling

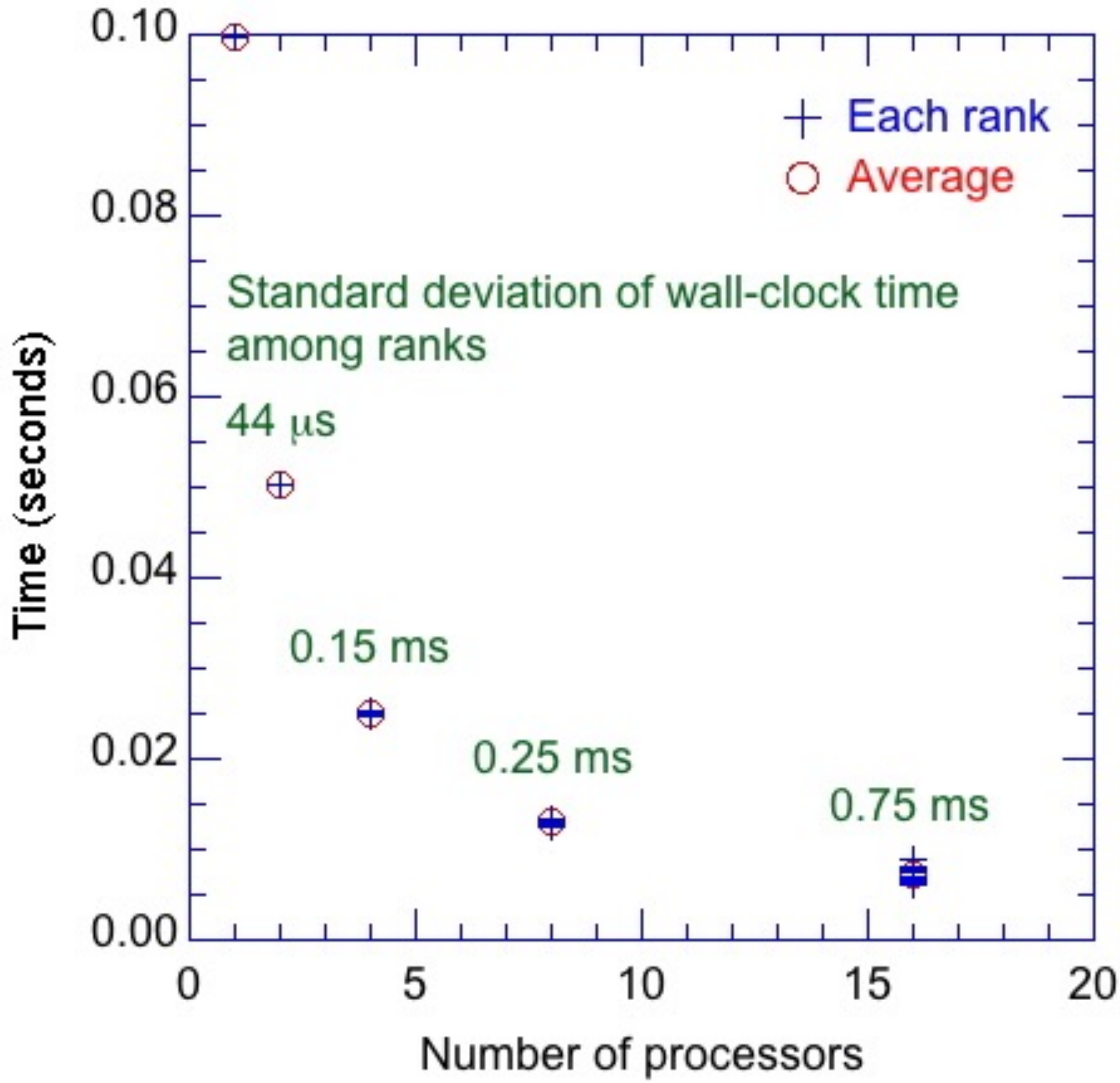
- **Speedup model:** $E_P = \frac{S_P}{P} = \frac{1}{1 + \frac{\beta}{\alpha}\frac{P\log P}{N}}$



Computation ~ ns
Communication ~ ms

global_pi.c: $N = 10^7$, on HPC (predecessor of Discovery)

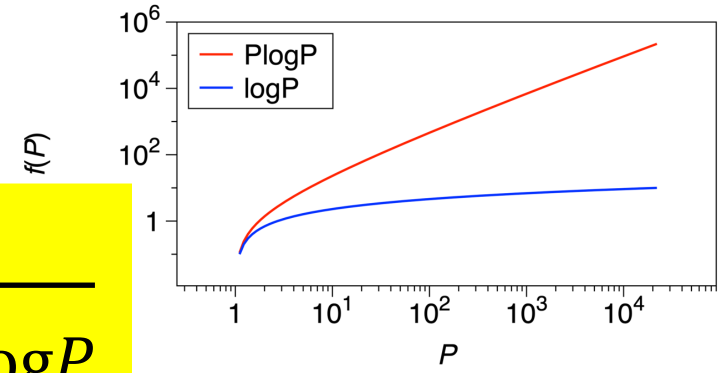
Runtime Variance among Ranks

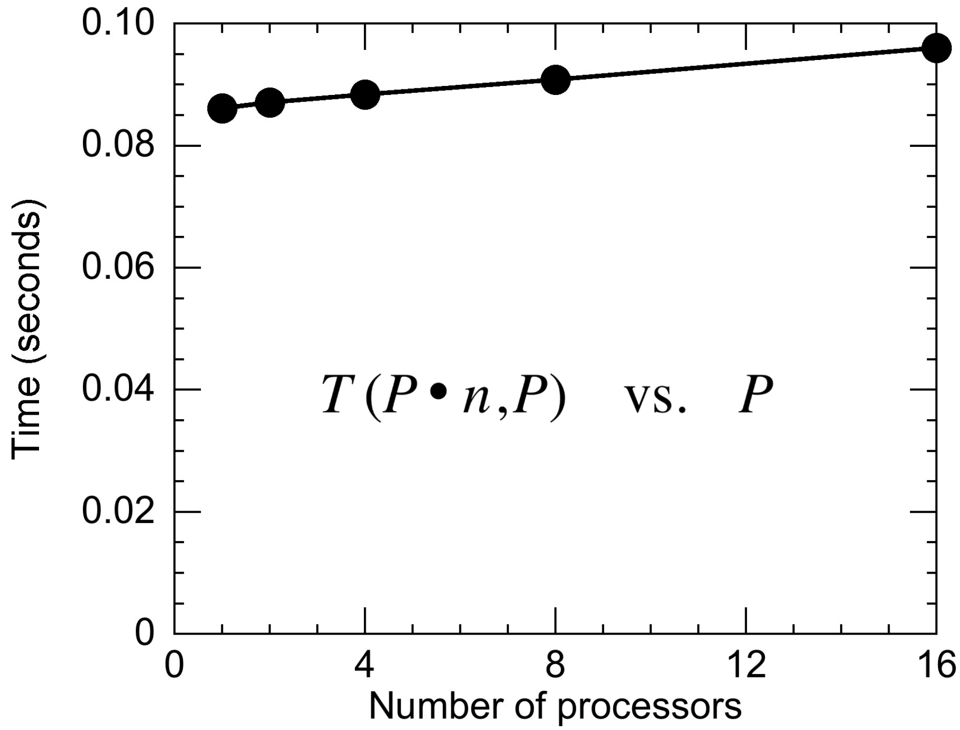


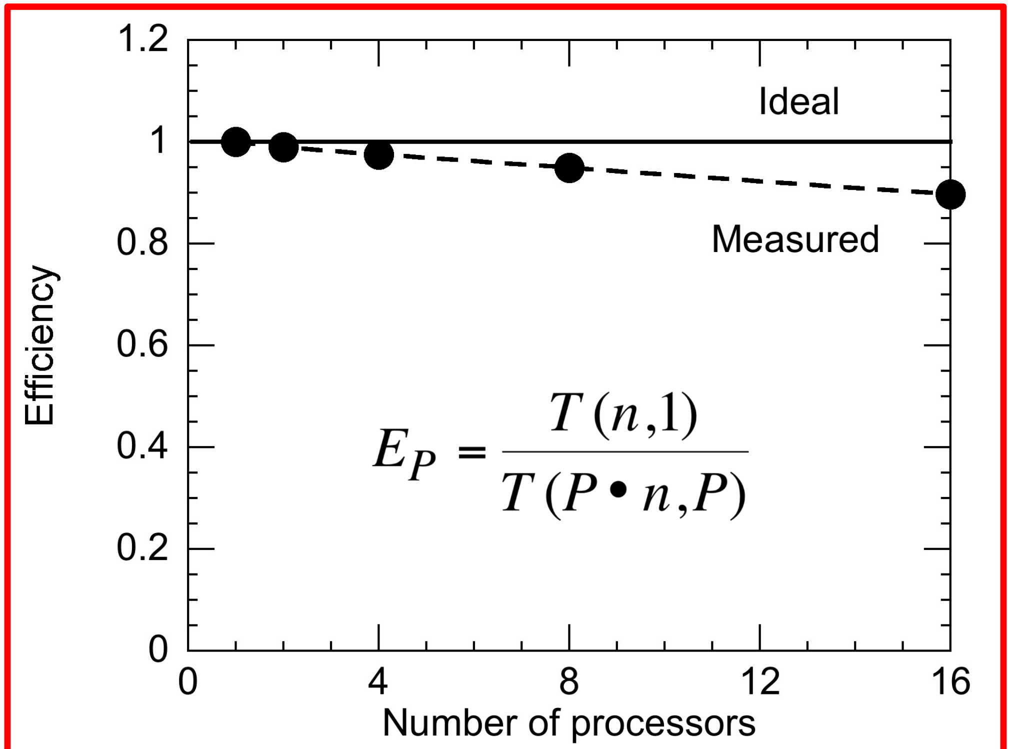
Isogranular (Weak) Scaling

- $n = N/P =$ **constant**
- **Efficiency:**

$$E_P = \frac{T(n,1)}{T(nP,P)} = \frac{\alpha n}{\alpha n + \beta \log P} = \frac{1}{1 + \frac{\beta}{\alpha n}\log P}$$







global_pi_iso.c: $N/P = 10^7$, on Discovery

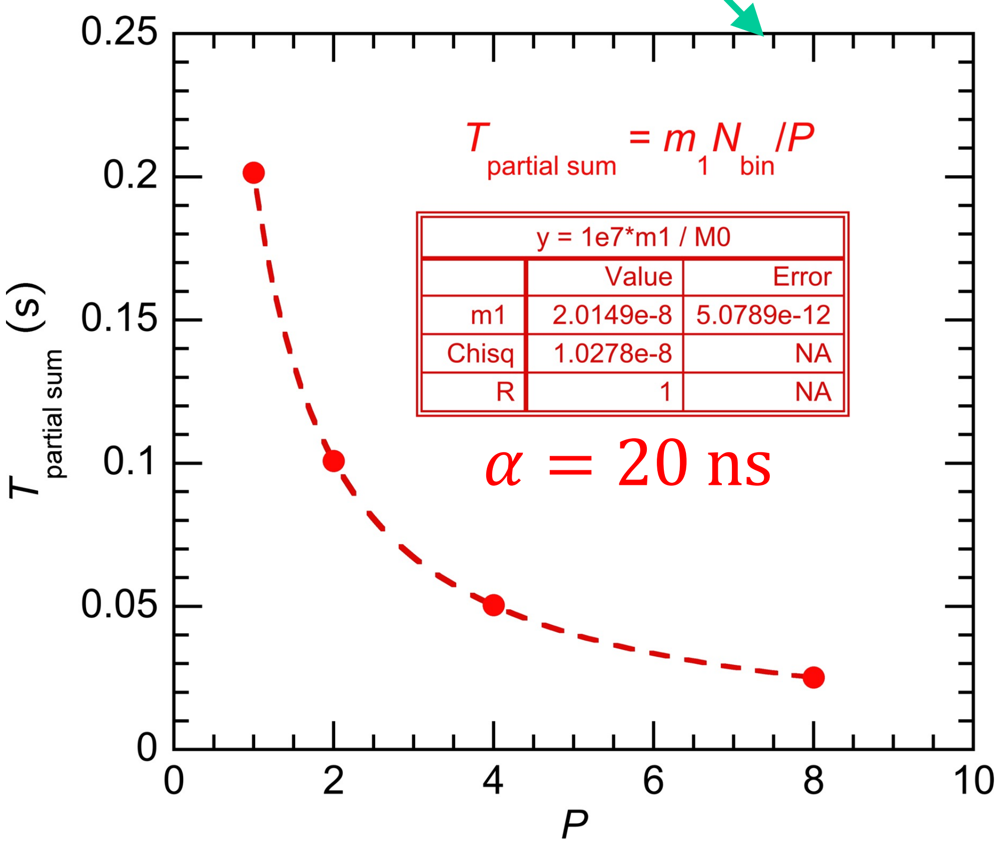
Fitting Fixed Problem-Size Scaling

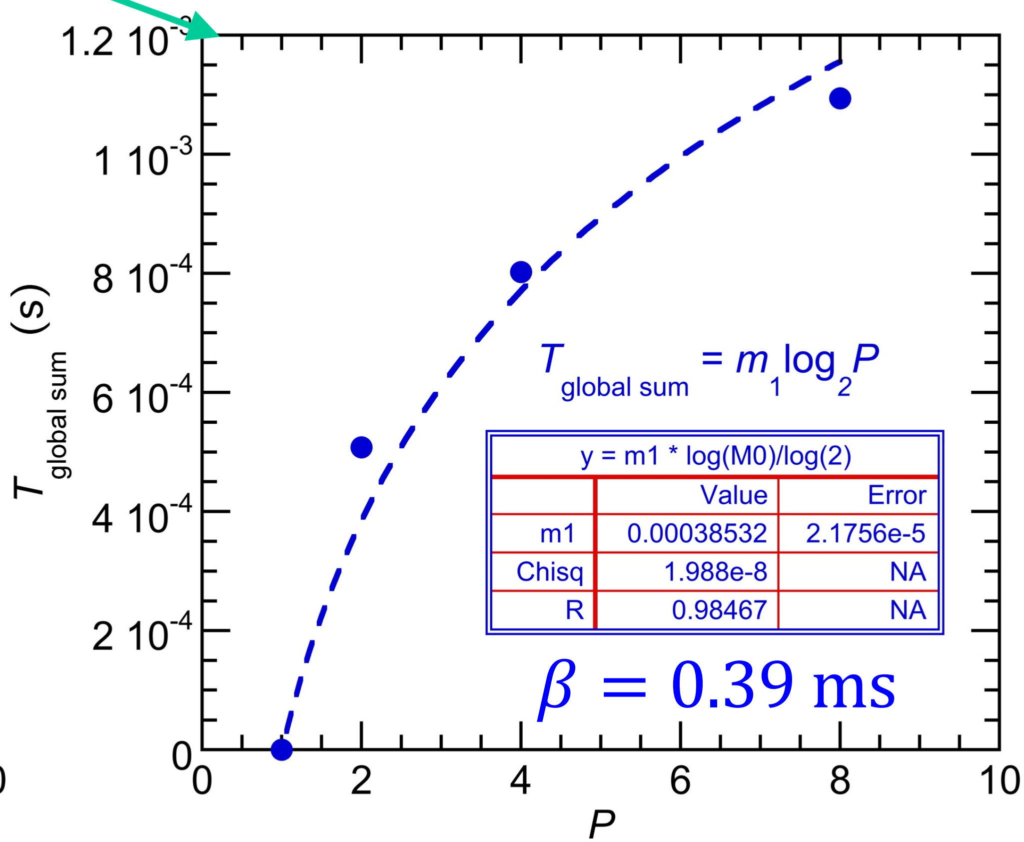
$$T(N,P) = \alpha\frac{N}{P} + \beta\log_2 P$$

cf.
[discovery]$ ping hpc-transfer.usc.edu
time=0.046 ms

```
for(i=myid;i<N;i+=P){...}
```

```
pi=global_sum(partial);
```





Computation ~ ns **Communication ~ ms**

global_pi_breakdown.c: $N = 10^7$, on 3.0 GHz Xeon

Scalability Analysis: Example

Computer Physics Communications 219 (2017) 246–254

A derivation and scalable implementation of the synchronous parallel kinetic Monte Carlo method for simulating long-time dynamics

Hye Suk Byun [a], Mohamed Y. El-Naggar [a,b,c], Rajiv K. Kalia [a,d,e,f], Aiichiro Nakano [a,b,d,e,f,*], Priya Vashishta [a,d,e,f]

$$T(N,P) = T_{\mathrm{comp}}(N,P) + T_{\mathrm{comm}}(N,P) + T_{\mathrm{global}}(P) = aN/P + b(N/P)^{2/3} + c\log P. \tag{6}$$

For isogranular scaling, the number of atoms per processor, $N/P = n$, is constant, and the isogranular parallel efficiency is

$$E_P = \frac{T(n,1)}{T(nP,P)} = \frac{an}{an + bn^{2/3} + c\log P} = \frac{1}{1 + \frac{b}{a}n^{-1/3} + \frac{c}{an}\log P}. \tag{7}$$

For fixed problem-size scaling, the global number of hemes, N, is fixed, and the speedup is given by

$$S_P = \frac{T(N,1)}{T(N,P)} = \frac{aN}{aN/P + b(N/P)^{2/3} + c\log P} = \frac{P}{1 + \frac{b}{a}\left(\frac{P}{N}\right)^{1/3} + \frac{c}{a}\frac{P\log P}{N}}, \tag{8}$$

and the parallel efficiency is

$$E_P = \frac{S_P}{P} = \frac{1}{1 + \frac{b}{a}\left(\frac{P}{N}\right)^{1/3} + \frac{c}{a}\frac{P\log P}{N}}. \tag{9}$$