



Yet Another Parallel Computations in *Mathematica*

Yuko Matsuda (matsuda@symbolics.jp)

Symbolic Systems LLC

Dean Dauger

Dauger Research, Inc.

Zvi Tannenbaum

Advanced Cluster Systems LLC

Abstract

Wolfram provides parallel computing functions built in *Mathematica*.

We have another type of parallel computing environment based on SEM(Supercomputing Engine for *Mathematica*). SEM connects *Mathematica* and MPI(Message Passing Interface), which is de facto standard parallel library in HPC world.

All functions of SEM are implemented as a *Mathematica* function, so *Mathematica* smoothly accesses MPI via these functions.

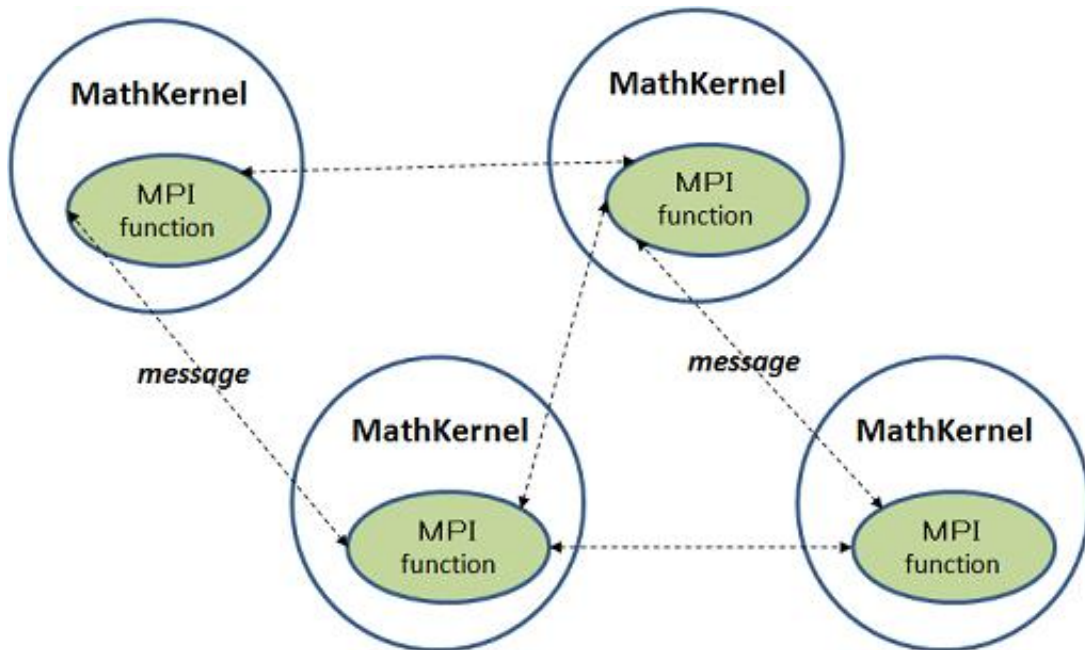
SEM has two advantages compared to *Mathematica* built-in parallel functions: one is the computational performance of *Mathematica*-SEM and the second one is the flexibility of communication topology.

In this paper we introduce the informal semantics of SEM and demonstrate several simulation samples in *Mathematica*/SEM.

MPI Programming with SEM

Message Passing Parallelism

Under the philosophy of MPI, the same program runs on all processes, which are MathKernels in the case of *Mathematica*, and each process exchanges a message which carries information and also controls the timing of synchronization among independently running parallel processes.



Mathematica hides this kind of parallelism in the back of built-in ParallelFunctions. On the other hand SEM provides explicit message passing functions as mpiFunctions. Explicit parallelism may lead to a tedious coding, however real world applications need complicated communication topology and should be expressed in explicit parallel languages like MPI.

How MPI Program Works with *Mathematica*/SEM

The global variable \$NProc returns the number of processes being available for computing, which equals to the number of MathKernels being currently available.

```
$NProc
```

```
8
```

MPI provides a process pool, so-called communicator. The communicator specifies to which pool each process belongs to. The current communicator is specified with the variable mpiCommWorld.

```
mpiCommWorld
```

```
0
```

And each process has its own Id, indicated with \$IdProc. Note that the Id starts at 0 in MPI.

```
$IdProc
```

0

The following expression `a=1` seems to simply assign 1 to `a`. However, the same expression runs on whole processes.

```
a = 1;
```

So you can see the value of the variable `a` in each process with `mpiGather`. `mpiGather` collects `{$IdProc,a}` in each process of `mpiCommWorld` into a variable `L` in the process0.

```
mpiGather[{$IdProc, a}, L, 0, mpiCommWorld];
Print[L]
```

```
{0, 1}, {1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1}, {6, 1}, {7, 1}
```

This function can be expressed more simply. `mpiCommWorld` and the process0 are suppressed in the following example.

```
mpiGather[{$IdProc, a}]
```

```
{0, 1}, {1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1}, {6, 1}, {7, 1}
```

We can control an operation to specific processes. The following program shows the value of `a` in the process3 has been changed from 1 to 2.

```
If[$IdProc == 3, a = 2];
mpiGather[{$IdProc, a}]
```

```
{0, 1}, {1, 1}, {2, 1}, {3, 2}, {4, 1}, {5, 1}, {6, 1}, {7, 1}
```

MPI functions run on whole processes, so `Clear[a]` will make clear the value of `a` in each process.

```
Clear[a];
mpiGather[a]
```

```
{a, a, a, a, a, a, a, a}
```

Send / Receive a Message

We demonstrate a more realistic example, calculating the area specified with `Sin[x]`, `{x,a,b}`. The simple computation is to integrate `Sin[x]` with `a` and `b`.

```
a = 1; b = 4;
 $\int_a^b \text{Sin}[x] dx // N$ 
```

```
1.19395
```

The integration of computing area will be replaced with the summation of trapezoids. There should be enough number of trapezoids for gaining certain accuracy and the following example sets the number `n` to 1000. Function `trap` will compute subarea consisting of trapezoids, which starts at `local_a` and ends at `local_b`. `h` is the width of each trapezoid.

The program computes `trap` with `local_a`, `local_b`, `local_n` and `h` and set it to `p`. Then the root process receive the result `p` from each process and sums up into variable `sum`. The rest processes simply sends `p` to the root.

```

trap[a_, b_, n_, h_] := Module[{x, s, integral},
  integral = (Sin[a] + Sin[b])/2;
  x = a;
  Do[x = x + h; integral = integral + Sin[x], {i, 1, n - 1}];
  integral = integral*h; integral]

a = 1; b = 4;
n = 1000; (* number of trapezoids *)
h = (b - a)/n; (* trapezoid base width *)
n = n/$NProc; (* the number of trapezoids on each process *)
a = a + $IdProc*n*h; b = a + n*h; (* local_a and local_b for each process *)

p = trap[a, b, n, h]; (* note: computed on all the process with different a and b *)
sum = p;
If[$IdProc == 0,
  (* in the root process *)
  Do[mpiRecv[v, i, 1, mpiCommWorld]; sum = sum + v, {i, 1, $NProc - 1}],
  (* in the rest of processes *)
  mpiSend[p, 0, 1, mpiCommWorld]]
If[$IdProc == 0, Print[N[sum]]]

```

1.19395

mpiRecv has a receiving buffer v, the id of target process (here i), the message tag (here 1) and the communicator mpiCommWorld. mpiSend has the same ones except a sending buffer p. mpiRecv and mpiSend will wait for the opponent's value to be placed in local communication buffer, so this kind of communication is called blocking communication.

Note. The latter part of program can be expressed as follows. mpiReduce is a reduction function.

```

p = trap[a, b, n, h];
mpiReduce[p, sum, mpiSum, 0, mpiCommWorld];
Print[N[sum]]

```

1.19395

Non-Blocking Communication

Blocking communication needs to wait for the opponent to put a value in local buffer and occasionally lose the chance that supposed idle processes can proceed another computations. So MPI provides a non-blocking communication and also SEM implements this as mpiSend and mpiRecv (note: 'I' indicates immediate).

The following program demonstrates a non-blocking communication in the ring topology where each process passes a value (x here) to the next process simultaneously. The first part of program allocates a certain size of data in each process as a variable x. The latter part performs the non-blocking communication and wait for the communication to finish.

The non-blocking communication needs to complete in the future. The send/receive status (sreq and rreq in the program) will be used in mpiWait in certain timing. mpiWait will wait for request status to be True.

```

successor := Mod[($IdProc + 1), $NProc];
predecessor := Mod[($IdProc - 1 + $NProc), $NProc];
blocksize = 1024;
x := Range[$IdProc*blocksize + 1, ($IdProc + 1)*blocksize];
Print[mpiGather[{$IdProc, Total[x]}]]

```

```

{{0, 524800}, {1, 1573376}, {2, 2621952}, {3, 3670528},
 {4, 4719104}, {5, 5767680}, {6, 6816256}, {7, 7864832}}

```

```

Do[
  mpiISend[x, successor, 1, mpiCommWorld, sreq];
  mpiIRecv[v, predecessor, 1, mpiCommWorld, rreq];
  mpiWait[sreq];
  mpiWait[rreq],
  {i, 0, $NProc - 1}]
Print[mpiGather[{$IdProc, Total[v]}]]

```

```

{{0, 7864832}, {1, 524800}, {2, 1573376}, {3, 2621952},
 {4, 3670528}, {5, 4719104}, {6, 5767680}, {7, 6816256}}

```

Sometimes we want to have an initiative to manage parallel computations. SEM provides `mpiWaitall`, `mpiWaitany` and `LoadBalanceFunction*` for this purpose.

Distributed Computing

Mathematica provides the parallel version of `Table[]`, `ParallelTable`. SEM also provides the same name function, however the semantics of both is different. *Mathematica*'s built-in parallel functions aims to support easy-to-use parallelism and the semantics of `ParallelTable` of *Mathematica* reflects this philosophy. Compare the following executions of the same program.

```
v = ParallelTable[i, {i, 1, 64}] (* built-in Mathematica *)
```

```

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64}

```

```
v = ParallelTable[i, {i, 1, 64}] (* SEM version *)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

The result of the SEM version of `ParallelTable` might not be happy because the only sub part of the whole result has returned (note: surely other processes has also other part of the result at `v` according to the process number). Usual parallel programming, especially distributed computing, expects data to be located in each process not in the root. The whole data will be needed at the last time of parallel computation and there should be usually several parallel computation steps from the start to the end. So the semantics of `ParallelTable` of SEM reflects to this designing philosophy.

The following program illustrates a parallel inner product with SEM's `ParallelTable`. The variables `x` and `y` keep corresponding sublist of random integers according to each process id.

```

serialDot[x_, y_] := Total[x*y];
RandomSeed[Prime[$IdProc + 1]];
x = ParallelTable[RandomInteger[{1, 10}], {i, 1, 32}];
y = ParallelTable[RandomInteger[{1, 10}], {i, 1, 32}];
mpiReduce[serialDot[x, y], sum, mpiSum, 0, mpiCommWorld];
Print[sum]

```

```
1119
```

```

(* here the operation "." is InnerProduct *)
Flatten[mpiGather[x], 1].Flatten[mpiGather[y], 1]

```

```
1119
```

All functions heading the name of `Parallel` are implemented with SEM's `mpi`-functions. So they have the same semantic philosophy as the same as `ParallelTable`.

Interactive MPI

Ordinary MPI program can be checked after being compiled together with Fortran or C++. MPI in *Mathematica* works interactively, which leads easy-to-understand parallel programming. This is a great advantage of SEM with the composition of performance.

Parallel Programming of Game of Life

Serial Version

Very popular program for Game of Life is constructed based on cellular automata's idea. *Mathematica* ver.7 has a powerful function CellularAutomaton and performs the below algorithm more efficiently, however the old style is adopted because of clearness of the algorithm.

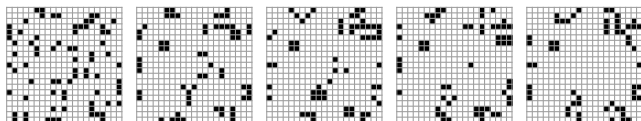
```
brdSize = 21;
brd = Partition[Table[If[Random[] > 0.8, 1, 0], {brdSize*brdSize}], brdSize];
(* random generated Life pattern in which 80% in average lives created *)
```

```
showBoard[brd_, opts : OptionsPattern[]] := ArrayPlot[brd, FilterRules[{Mesh -> True, opts}], Options[ArrayPlot]]
```

```
LifeGame[config_, t_] := Module[{livingNghbrs, deadAliveCheck, mat, update, gather},
  (* deadAliveCheck[the status of the cell, the number of lives] *)
  deadAliveCheck[1, 2] := 1;
  deadAliveCheck[_ , 3] := 1;
  deadAliveCheck[_ , _] := 0;
  Attributes[deadAliveCheck] = Listable;
  (* calculating the number of lives surrounding each cell *)
  livingNghbrs[mat_] := Plus @@ (RotateRight[mat, #1] &) /@ {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}};
  (* repeat to update each status of cells with specified times t *)
  update[mat0_] := deadAliveCheck[mat0, livingNghbrs[mat0]];

  NestList[update, config, t]]
```

```
showBoard /@ LifeGame[brd, 4]
```



Parallel Version

The parallel algorithm runs as follows:

- (1) Scatter the whole cells into each process with `mpiScatter`. The scattered one is placed on `m`.
- (2) Add null row spaces to the edge of `m` because these places will be used as a boarder of each partitioned board. This is performed with padding.
- (3) Repeat the following steps with loop number of `t`:
 - (3-1) Exchange border cells among adjoining processes with `EdgeCell` (this also SEM's function).
 - (3-2) Update `m` with the rule `deadAliveCheck` at each process with update
- (4) Finally collect local `m` with `mpiGather` and peel the border of each `m` then rebuild the board with the original size.

Note that in nesting Function the local variable `a` is not used. This is because `a` does not contribute to the computation. However it will be needed when we implement `FixedPoint` in similar kind of problem to check to notify that the old one and the new one ARE the same.

```
padding[mat_] := Module[{m}, m = Table[0, {Length[mat] + 2}, {Length[mat][[1]]}];
  m[[2 ;; -2]] = mat;
  m];
peal[mat_] := Module[{m, n},
  {m, n} = Dimensions[mat];
  mat[[2 ;; m - 1]]];
parallelLifeGame[config_, t_] := Module[{livingNghbrs, deadAliveCheck, mat, update},
  deadAliveCheck[1, 2] := 1;
  deadAliveCheck[_ , 3] := 1;
  deadAliveCheck[_ , _] := 0;
  Attributes[deadAliveCheck] = Listable;
  livingNghbrs[mat_] := Plus @@ (RotateRight[mat, #1] &) /@ {{-1, -1}, {-1, 0},
  {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}};
  update[mat0_] := deadAliveCheck[mat0, livingNghbrs[mat0]];

  (* parallel computation part *)
  mpiScatter[config, mat, 0, mpiCommWorld];
  mat = padding[mat];
  Nest[Function[{a}, EdgeCell[mat]; mat = update[mat]], mat, t];
  Flatten[peal /@ mpiGather[mat], 1]
]
```

Timing Comparison between the Serial and the Parallel Version

The time consumed for the management of remote computing is almost zero, however the cost of data gathering is not negligible. So to gain a good performance in parallel computation, certain size of computation space is needed. In the following case the board will be set 800*800. This class of size can show a good performance, almost 8 times faster gained.

```
$NProc

8

brdSize = 800;
brd = Partition[Table[If[Random[] > 0.8, 1, 0], {brdSize*brdSize}], brdSize];

Timing[LifeGame[brd, 10];] (* this time Nest is used insted of NestList *)

{9.73398, Null}

Timing[parallelLifeGame[brd, 10];]

{1.97058, Null}
```

Supporting Dynamic Data Relocation

Real parallel programs frequently need data exchange among processes. However to write such a program with mpi functions is not so easy because of considering the remapping algorithms. SEM provides two kinds of data remapping functions EdgeCell and ElementMange for this purpose. These two functions were born in the background of being SEM having been applied in real world applications.

EdgeCell

EdgeCell is used in the parallel version of Game of Life. The following example illustrates the mechanism of this function, exchanging the row-end boards between adjoining processes.

```
m = Partition[Range[16], 4] + $IdProc*10;
MatrixForm[Transpose[#]] & /@ mpiGather[m]
```

$$\left\{ \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}, \begin{pmatrix} 11 & 15 & 19 & 23 \\ 12 & 16 & 20 & 24 \\ 13 & 17 & 21 & 25 \\ 14 & 18 & 22 & 26 \end{pmatrix}, \begin{pmatrix} 21 & 25 & 29 & 33 \\ 22 & 26 & 30 & 34 \\ 23 & 27 & 31 & 35 \\ 24 & 28 & 32 & 36 \end{pmatrix}, \begin{pmatrix} 31 & 35 & 39 & 43 \\ 32 & 36 & 40 & 44 \\ 33 & 37 & 41 & 45 \\ 34 & 38 & 42 & 46 \end{pmatrix} \right\},$$

$$\left\{ \begin{pmatrix} 41 & 45 & 49 & 53 \\ 42 & 46 & 50 & 54 \\ 43 & 47 & 51 & 55 \\ 44 & 48 & 52 & 56 \end{pmatrix}, \begin{pmatrix} 51 & 55 & 59 & 63 \\ 52 & 56 & 60 & 64 \\ 53 & 57 & 61 & 65 \\ 54 & 58 & 62 & 66 \end{pmatrix}, \begin{pmatrix} 61 & 65 & 69 & 73 \\ 62 & 66 & 70 & 74 \\ 63 & 67 & 71 & 75 \\ 64 & 68 & 72 & 76 \end{pmatrix}, \begin{pmatrix} 71 & 75 & 79 & 83 \\ 72 & 76 & 80 & 84 \\ 73 & 77 & 81 & 85 \\ 74 & 78 & 82 & 86 \end{pmatrix} \right\}$$

```
EdgeCell[m];
MatrixForm[Transpose[#]] & /@ mpiGather[m]
```

$$\left\{ \begin{pmatrix} 79 & 5 & 9 & 15 \\ 80 & 6 & 10 & 16 \\ 81 & 7 & 11 & 17 \\ 82 & 8 & 12 & 18 \end{pmatrix}, \begin{pmatrix} 9 & 15 & 19 & 25 \\ 10 & 16 & 20 & 26 \\ 11 & 17 & 21 & 27 \\ 12 & 18 & 22 & 28 \end{pmatrix}, \begin{pmatrix} 19 & 25 & 29 & 35 \\ 20 & 26 & 30 & 36 \\ 21 & 27 & 31 & 37 \\ 22 & 28 & 32 & 38 \end{pmatrix}, \begin{pmatrix} 29 & 35 & 39 & 45 \\ 30 & 36 & 40 & 46 \\ 31 & 37 & 41 & 47 \\ 32 & 38 & 42 & 48 \end{pmatrix} \right\},$$

$$\left\{ \begin{pmatrix} 39 & 45 & 49 & 55 \\ 40 & 46 & 50 & 56 \\ 41 & 47 & 51 & 57 \\ 42 & 48 & 52 & 58 \end{pmatrix}, \begin{pmatrix} 49 & 55 & 59 & 65 \\ 50 & 56 & 60 & 66 \\ 51 & 57 & 61 & 67 \\ 52 & 58 & 62 & 68 \end{pmatrix}, \begin{pmatrix} 59 & 65 & 69 & 75 \\ 60 & 66 & 70 & 76 \\ 61 & 67 & 71 & 77 \\ 62 & 68 & 72 & 78 \end{pmatrix}, \begin{pmatrix} 69 & 75 & 79 & 5 \\ 70 & 76 & 80 & 6 \\ 71 & 77 & 81 & 7 \\ 72 & 78 & 82 & 8 \end{pmatrix} \right\}$$

ElementManage

The following code is borrowed from a plasma simulation program written in *Mathematica*/SEM. The original data is hold in the variable data. ParticleManagePicker works as a switch for data reallocation according to the value of `data[[i]][[1]][1]` and another factors nx and nproc.

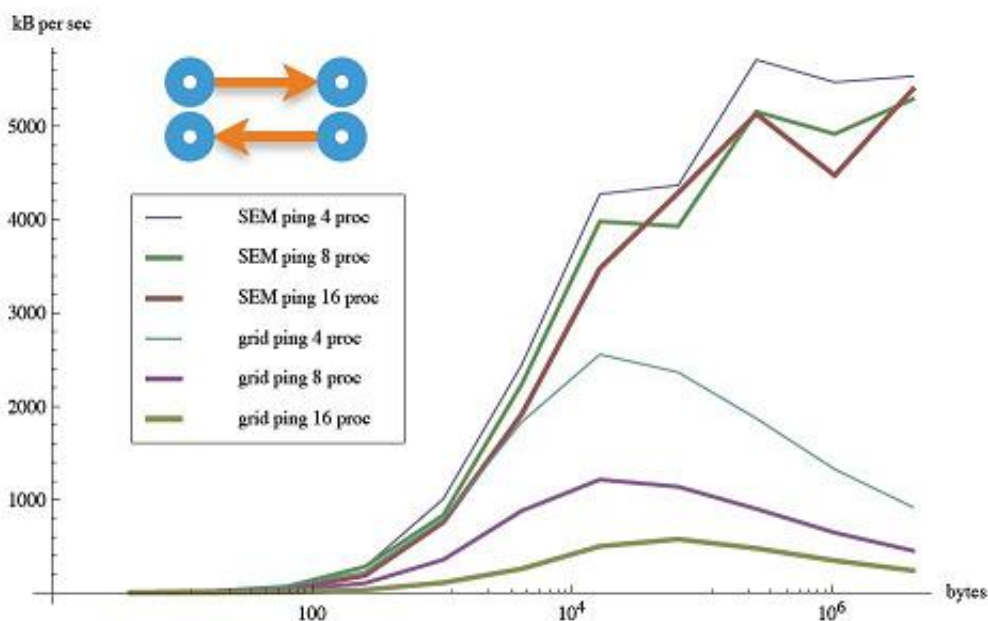
```
data = {
  {{1.85393, 5.09467}, {-0.10203, -0.0308564}},
  {{0.828114, 10.3604}, {0.0497702, 0.119969}},
  {{1.85283, 29.9882}, {-0.0263, 0.0893992}},
  {{2.07997, 30.4864}, {0.0540052, -0.0990736}},
  {{3.54256, 20.7065}, {0.0561277, 0.0690143}},
  {{1.62291, 29.3279}, {0.0965776, 0.00641346}},
  {{2.73342, 0.691632}, {-0.0684564, -0.0585249}},
  {{3.43298, 15.3105}, {0.0705268, -0.0850361}},
  {{0.836714, 30.42}, {0.026655, 0.115649}}
};
nx = 4; nproc = 8;
ParticleManagePicker[in_] := Mod[(in[[1]][1]*nproc)/nx, nproc];
out = ElementManage[data, ParticleManagePicker]; mpiGather[{$IdProc, Union[out]}]
// Transpose // TableForm
```

0	1		2	3		4		5		6
				1.62291	0.0965776					
	0.828114	0.0497702		29.3279	0.00641346					
	10.3604	0.119969		1.85283	-0.0263	2.07997	0.0540052	2.73342	-0.0684564	3.4329
	0.836714	0.026655		29.9882	0.0893992	30.4864	-0.0990736	0.691632	-0.0585249	15.310
	30.42	0.115649		1.85393	-0.10203					
				5.09467	-0.0308564					

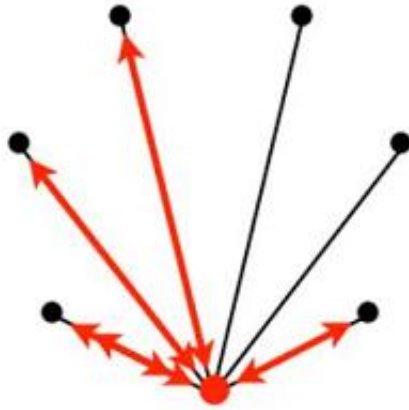
Performance Comparison between *Mathematica* Built-in Parallel Functions and SEM Functions

The following chart illustrate the performance difference with swap program and ping pong program between *Mathematica* and *Mathematica*/SEM. The swap program swaps data between the pair of adjoining processes and the ping pong program sends a data then receives the same data between the pair of adjoining processes. The horizontal axis indicates the size of transmitted data and the vertical axis indicates the bandwidth of data transmission.

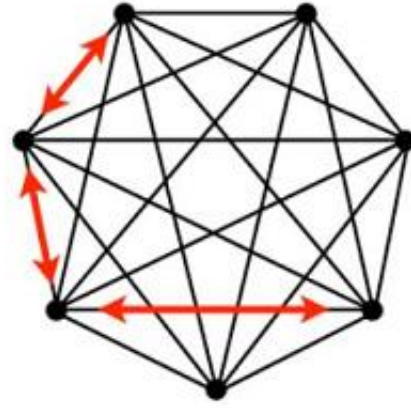
Saturated curves after 10^4 byte point are of *Mathematica*.



The main reason that the performance of *Mathematica* will saturate after the certain point passed is simple. Parallel communication topology adopted by *Mathematica* is the master-slave or tree structure illustrated in the left side below. The communication overhead in the root process will be very tough. On the other hand SEM fundamentally supports direct communication among processes(see the right diagram below). This topology can simulate any kind of communication topology.



gridMathematica's topology



SEM's topology

Parallel Table performance comparison test at SEM1.2 and MacOS10.6.2 4-core/8-kernel

SEM

```
RandomSeed[Prime[$IdProc + 1]];
Timing[a = ParallelTable[RandomInteger[{1, 1000}], {100 000}];]
{0.00153, Null}

Timing[mpiGather[a];]
{0.104179, Null}

Timing[ParallelTable[RandomInteger[{1, 1000}], {100 000}, 0];]
{0.110467, Null}
```

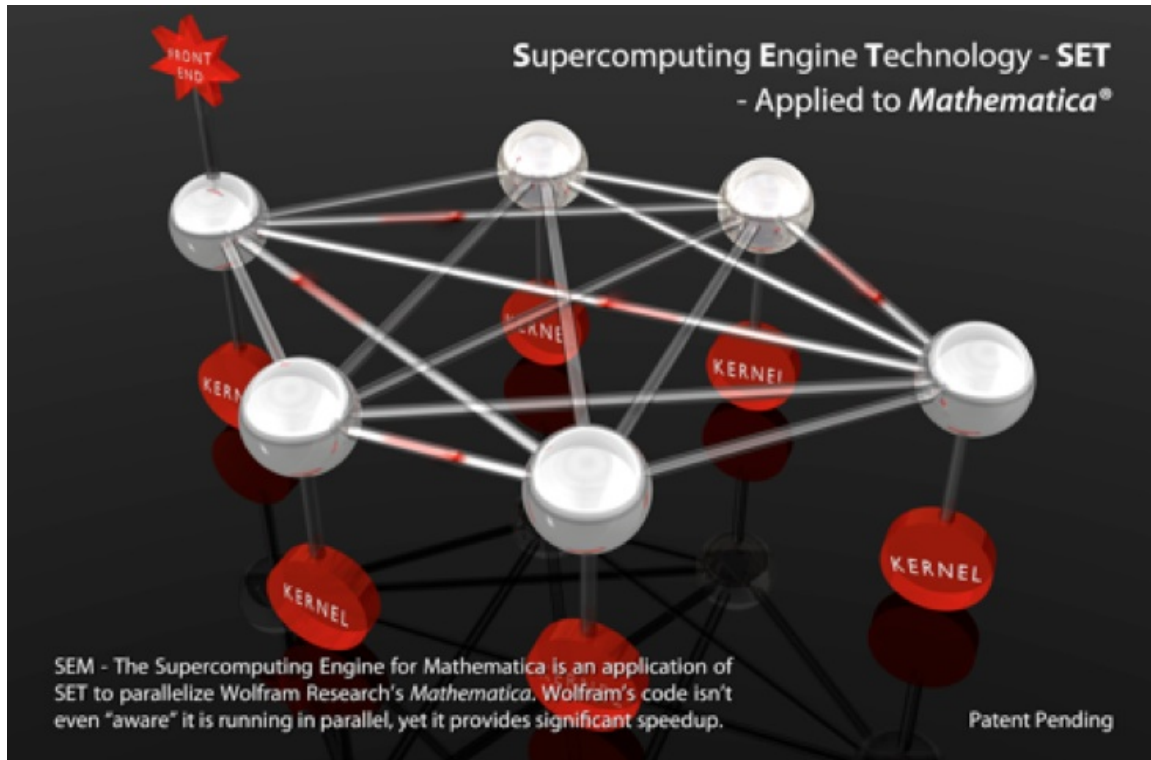
Mathematica built-in

```
CloseKernels[]; LaunchKernels[8];

Timing[ParallelTable[RandomInteger[{1, 1000}], {100 000}];]
{0.566442, Null}
```

Structure of SEM

SEM is one of SET(Supercomputing Engine Technology) applications, which intercept and add several libraries like MPI between the front-end and kernels in kernel base languages.



In the case of SEM the red kernel indicates a MathKernel and the white one does a process which connect inner kernels and an outer system like MPI.

Conclusions

SEM supports the subset of openMPI, not a full set. Even so, the following functions are enough powerful to write real world parallel programs.

Collective functions: `mpiGather`, `mpiAllgather`, `mpiScatter`, `mpiAlltoall`, `mpiBcast`, `mpiReduce`,...

Parallel function for matrix operations: `ParallelTranspose`, `ParallelProduct`, `ParallelTr`, `ParallelIdentity`, `ParallelOuter`, `ParallelInverse`, ...

Parallel control: `ParallelFunction`, `ParallelDo`, ...

more detail: see <http://www.symbolics.jp/>

