Application Characteristics of Point Designs

1 Introduction

The Application Groups at PAWS '96 posed a set of questions to the point design teams that were designed to quantify the differences and relative capabilities of the designs. We received five responses that revealed, as we will see, as much about the questions as the machines! So, we present this section partly as a contribution to a good set of metrics and discriminators for evaluating diverse architectures.

The text presents the original set of questions, which are grouped into eight delineating broad features, and one (question #9 with six parts) with a set of simple kernels for which one is asked to analyze expected performance. We present and analyze the response of:

Point Group 1. MORPH: 256 Teraflop Peak

Point Group 4. GRAPE: 1 Petaflop Peak

Point Group 6. HPAM: 1 Petaflop Peak

Point Group 7. I-ACOMA: 1 Petaflop Peak

Point Group 8. NJIT: 100 Teraflop Peak

For clarity, we have sometimes also given scaled numbers in response to a uniform 1 Petaflop peak performance. Below, we make some broad remarks on the review and then follow with specific remarks for each section, going through questions 1, 3, 9, 2, 4–8 in an order that links benchmarks with detailed machine parameters.

1. Estimated Cost

The question asked — use the SIA estimate that semiconductor hardware will be 25 times cheaper per transistor in the 2006 time frame than it is in 1996. You may take $50 per Megabyte for 1996 memory and $5 per Megaflops per second for 1996 sustained RISC processor performance (which roughly scales with transistor count), or else otherwise justify your estimate.

General Remarks

It was correctly noted by some responders that $5 per sustained Megaflops per second (quoted in question) was not so easy to use as "sustained" depends on application analysis, and easier is processor chip cost. We changed cost estimates to $2,000 a processor chip and $0.1 per Megabytes of memory in 2007. Note: 1994 SIA projections are $80 for a volume microprocessor chip and $0.04 per Megabyte for volume DRAM in the year 2007 (this is $80 per a single two-Gigabytes memory DRAM chip). In the table below, we used this uniform estimate, and not that given by Point Groups who made varied assumptions.

The high cost per processor includes estimates of low volume, packaging and connection costs. The DRAM is assumed to be a little more than SIA volume cost, as used near start of introduction when price will be higher than volume cost. Note that the wide variation of total Petaflop cost recorded below largely reflects different total memory assumptions. This reflects

- lack of clarity as to application needs, and
- different point group cost assumptions for the basic (two Gigabytes) memory chips, motivating different memory sizes.

Team 1: MORPH

Memory:	20K	2 Gigabyte Memory Chips	$ 4 M
CPU:	8K	Processor	$ 16 M
		Total (256 Teraflops)	$ 20 M
		Total (1 Petaflop)	$ 80 M

Team 4: GRAPE

		This aims at a Petaflop in 5 years with special purpose chips:	
		10K chips at $1 K per chip	$ 10 M

Team 6: HPAM

		Lowest Level (L5)	
	8K	2 Gigabyte Memory Chips modified for PIM ($3K each)	$ 24 M
		Multiply by 2.5 for other levels	$ 60 M

Team 7: I-ACOMA

Memory:	50K	2 Gigabyte Memory Chips	$ 10 M
CPU:	20K	Processors	$ 40 M
		Total	$ 50 M

Team 8: NJIT

Memory:	160K	2 Gigabyte Memory Chips	$ 32 M
CPU:	10K	Processors	$ 20 M
		Total (100 Teraflops)	$ 52 M
		Total (1 Petaflop)	$ 204 M
		Compare with NJIT Team 100 Teraflops Estimate without memory/processor breakup	$ 50 M

3. Hardware Specifications

Here the question asked for peak performance rates of processors, sizes and structures of the various levels of the memory hierarchy (including caches), latency and bandwidth of communication links between each level, etc. The answers here were difficult to collate—partly because different architectures are naturally described differently, but partly because we were not very precise in the question. We record first the processor structure, and then the memory hierarchy.

Processors and Their Characteristics

Note that base chip is typically 1 GHz with performance varying from 10 Gigaflops (Team 8) to 64 Gigaflops (Team 7). This represents uncertainty in technology extrapolation for the basic high-performance microprocessor.

Team 1: MORPH (256 Teraflops)

8K, 1 GHz Processor chips each containing four or eight C.P.U.'s with a peak performance of 32 Gigaflops per chip.

Team 4: GRAPE

10K Specialized Processor chips performing at 90 Gigaflops/chip

Team 6: HPAM

This has five levels

Level	# Processors	Performance per Processor Gigaflops	Performance per Level Teraflops
L1	1	200	0.2
L2	712	100	71
L3	2,136	70	150
L4	5,696	50	285
L5	14,240	35.3	500
Total	22,785	——	1 Petaflop

Team 7: I-ACOMA

Processor chip is eight C.P.U.'s (or threads), each issuing eight instructions per 1 GHz cycle. Sustained performance of 50 Megaflops, peak 64 Megaflops. 20K processor chips, packaged as 2,000 nodes. Each node is a 10 C.P.U. shared memory system.

Team 8: NJIT (100 Teraflops)

Processor chip is 1 GHz and peak performance is 10 Gigaflops. The base machine has 10,368 processors.

Memory Hierarchy

Here, there is even more variation in design and presentation. One remarkable architectural differences is the variation in the amount of memory on chip in the various designs. This varies by a factor of 1,000 from a few Megabytes (Team 7, 8) to $\sim$ 1.8 Gigabytes (Team 2, 6). The petakernels seem to favor large chip caches and it would seem that there may be an optimal design in between these extremes.

We found answers difficult to interpret unambiguously as the unit was often not clear—was the L1 cache per C.P.U. (of which there are many chips) or per chip!

We have devised a table, and inserted some material in it. This table—as we will see in Question 9—has data of relevance to analyzing performance, and encompasses all information presented by teams. However, no team clearly gave all the information!

The table has, in general, the following columns: memory subsystem, unit of subsystem, size per unit, total size per machine, page or cache line size, latency, and access bandwidth per unit (but we only record in each table for a given team those for which we have data).

Note that the highest level of memory often has two sets of characteristics—one viewed as memory per processor node—another viewed as the total distributed shared memory seen by all (other) nodes.

Team 1: MORPH
Memory Hierarchy 256 TeraFLOPS Machine

Memory Subsystem	Unit	Size per Unit	Total Size per Machine	Access Bandwidth per Unit
L1	? per CPU	8–16 KB	?	
L2	per processor chip	128 MB	1 Terabyte	90 Gigabytes per second per chip
Main Memory	per machine		40 Terabytes	This is cross-sectional bandwidth. 1.2 Gigabytes per second per chip.

Team 6: HPAM
Level 1 (fastest processor)

Memory Subsystem	Unit	Size per Unit	Total Size per Machine
L1	?	32 KB	32 KB
L2	per chip	1 MB	1 MB
L3	per chip		512 MB
Main Memory	?		

Team 6: HPAM
Level 5 (lowest PIM)

Memory Subsystem	Unit	Size per Unit	Total Size per Machine
Memory next to each CPU	per CPU (16 per chip)	0.11 GB	1.6 TB
On Chip Memory	per chip	1.8 GB	25.6 TB

Team 7: I-ACOMA

Memory Subsystem	Unit	Size per Unit	Total Size per Machine	Cache Line Size	Latency (nanoseconds)
L1	per chip or CPU	32 KB		64 B	1
L2 on chip	per chip	1 MB	20 GB	64 B	10
L3 off chip	per chip or node?	512 MB			50
(10-chip) Node Memory	per node	5 GB	100 TB		150
Main Memory		——	100 TB		> 1000

Team 8: NJIT
(100 Teraflops)

Memory Subsystem	Unit	Size per Unit	Total Size per Machine	Access Bandwidth per Unit
Cache	per chip	16 MB	0.16 TB	
Main Memory	per chip	32 GB	320 TB	30 Gigabytes per second per chip (incl. latency)
Card (8 processors)	per card	256 GB	320 TB	1.54 Gigabytes per second per chip
Main Memory	——	——	320 TB	a) with latency 1.075 Gigabytes per second per chip b) 6 Gigabytes per second DMA per chip c) Bisection Bandwidth 3 Gigabytes per second per chip

9. Quantitatively assess the performance of the proposed system on the following computational problems, based on design parameters and execution model. Justify the analysis. In the following, $n = 10^8$; arrays are presumed appropriately dimensioned.

a. SAXPY loop

```
do i = 1, n
   c(i) = 3.141592653589793 * a(i) + b(i)
enddo
```

b. Large-stride vector fetch and store

```
do i = 1, n
   b(131*i) = a(131313*i)
enddo
```

c. Irregular gather/scatter

```
!     This loop initializes the array idx with pseudo-random numbers
!     between 1 and n.
do i = 1, n
   idx(i) = 1 + mod (13131313*i, n)
enddo

!     Tested loop:
do i = 1, n
   b(idx(i)) = a(i)
enddo
```

d1. 3-D Jacobi kernel, typical of many 3-D physical modeling codes.

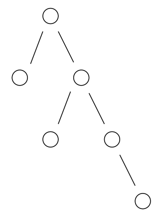
```
nc = 5; nx = 1000; ny = 1000; nz = 1000
do k = 1, nz
   do j = 1, ny
      do i = 1, nx
         do m = 1, nc
            do mp = 1, nc
               a(i,j,k,m) = u(mp,m) * (b(i+1,j,k,mp) + b(i-1,j,k,mp) &
                   b(i,j+1,k,mp) + b(i,j-1,k,mp) + b(i,j,k+1,mp) + &
                   b(i,j,k-1,mp)) + a(i,j,k,m)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

d2. Repeat the above calculation with the parameters

```
nc = 150; nx = 100; ny = 100; nz = 100
```

e. Processing of tree-structured data.

Consider a tree structure, such as



except with 10^4 nodes and an arbitrary structure, with one random integer at each node. Is this tree a subtree of a similar tree of size 10^9 nodes? Find the path to the node of the subtree in the large tree.

General Comments on Question 9 (Petakernels)

The current kernels are too small as they use such a sufficiently small amount of memory (Table A) that in many machines, all the data can be fitted in (one of) the caches. For instance, the vector applications 9a),b),c) should have a size $n = 10^{11}$ to get to Terabyte storage level.

Further, we need to define initial positioning of data. You get much better performance if you can assume data already in cache (natural if kernels iterated) than if you have to move. Fox suggests taking large kernels (which don't all fit in cache) but calculate performance with optimal positioning of data.

Table A: Summary of Six Petakernels: 9a) b) c) d1) d2) e)

a) 2×10^8 Double Precision Operations with memory needs of 2.4 Gigabytes. Time at one Petaflop is $0.2\,\mu$s if 10^{15} floating operations per second can be realized.

b) 2×10^8 Operations with memory need of 1.6 Gigabytes. Natural minimum time at one Petaflops is $0.2\,\mu$s. Of course, large stride implies that one expects to be dominated by fetching data, and so will "see" the cross-sectional bandwidth, and not C.P.U. performance.

c) Similar to b) with memory needs of 2.4 Gigabytes. Although irregular access will dominate (as no floating point operations in second loop), we will take $0.2\,\mu$s as "unit" performance. Note that first loop setting `idx` was correctly viewed as initialization by some point groups who only analyzed performance on second loop.

d1) This uses 0.08 Terabytes of memory (need to increase by a factor of at least 50 to get "petascale" memories by setting $n_x = n_y = n_z = 4,000$ perhaps) and $150\,\mu$s timing (for one Petaflop) optimal performance.

d2) This uses 2.4 Gigabytes of memory (need to increase by $n_x = n_y = n_z = 1,000$ to get a factor of 1,000 more memory), and has $135\,\mu$s timing for one Petaflop optimal performance. Notice that it is most natural to parallelize over n_x, n_y, n_z and leave m and mp loops "inside node" as these full matrix operations lead to most data re-use.

e) This problem is too small as in a simple case of a balanced tree, one only needs 10^5 operations to decide that a tree with 10^4 nodes is not contained in one with 10^9 nodes. We need a parallelism of at least 10^6 in any Petaflop machine to achieve maximal performance.

The algorithm needs to specify a little more about the tree to decide on the best algorithm. If integers at nodes are random, then one needs only to compare root of small tree with integers at nodes of large tree to reject a possible match! Suppose big tree was nine levels each with 10 children for 10^9 nodes, and little tree was a similar four-level branching of 10 data structure. Then we need only to compare root to nodes at top five levels of big tree. In general, performance depends on branching factors and how often one will get matches. "Random" integers will match rarely!

We give, in Table B, a summary of projected performance relative to naive optimum. Note that only three point groups gave a full set of numbers, and we ignored 9(e) as, although a nice test, it needs clarification. The projections of Team 7 were pessimistic, as they had poor cache locality, but noted (without details) that prefetching could improve. Team 8 gave no details on their projections. Note that we represent performance by a factor compared to a naive Petaflops machine, and that we scaled each machine to one Petaflop to get this factor, i.e. we removed differences in peak performances in this table.

Table B: Summary of Results — all machines scaled to one Petaflop

Kernel	Natural Petaflop Performance	Actual Performance Degradation Factor		
		MORPH	I-ACOMA	NJIT
9(a)	$0.2\,\mu$s	1	10^6	3.5
9(b)	$0.2\,\mu$s	400	10^7	10
9(c)	$\sim 0.2\,\mu$s	200	10^6	12
9(d1)	$150\,\mu$s	1	10^3	7
9(d2)	$135\,\mu$s	1	300	100

Team 1: MORPH Detailed analysis of question 9

- SAXPY Loop (for all kernels assumed Double precision—8 bytes)
  - Align a, b, and c, 2 loads, 1 store, 2 flops, prefetch data
  - Performance at full processor rate (data fits in L2 cache)
  - 8K * 32 GF → 256 TeraOps

- Large stride Vector fetch/store
  - Stream data at the processor chip limit/bisection BW limit (global references)
  - Load around caches using programmable logic, avoiding loading of useless data (e.g., rest of cache block)
  - 10 Terabytes per second over 8K processors = 1.2 Gigabytes per second per processor = 75 Megaflops per processor
  - 8K * 75 = 600 Gigaflops
  - $\rightarrow$ Can increase performance by buying more bisection bandwidth $$
- Irregular Gather/Scatter (ignore initialization loop)
  - Align a and idx, stream writes to memory system; performance limited by bisection BW
  - This time 1.2 Gigabytes per second per processor = 150 Megaflops per processor = 1.2 Teraflops
- 3D Jacobi Kernel (part 1)
  - Domain decomposition in i, j, k, total data requirement per processor fits in L2 cache, tile again for L1 cache (with prefetching)
  - Need special remap hardware to make 3D structure pack tightly into cache (configurable hardware)
  - Execute at close to peak, 8K * 32 Gigaflops = 256 TeraOps
- 3D Jacobi Kernel (part 2)
  - Loop interchange, parallelize m and mp loops (150 * 150 = 22,500 iterations), each does Jacobi over 1 M matrix element (i, j, k)
  - 1 M matrix fits in L2 cache, tile if necessary and prefetch into L1
  - Performance close to peak, 256 TeraOps
- Tree structure

  What the heck is a good algorithm for this? Can you control the mapping of the tree structure into memory (e.g., cdr-coding?);
  - Load around the cache for "big tree" to avoid loading useless parts of cache lines
  - Map tree by locality (span root at each level across successive machine bisections to limit BW requirement)
  - Replicate small tree several times to avoid bisection BW limitation, in fact it fits in the cache of the processors

- Global remote load limited 1/2 must cross bisection BW, so this means a peak of 300 M compares/proc (64 bit pointers)
- 8K * 300M = 2.4 Tera compares/sec

Team 4: GRAPE Detailed analysis of question 9 by Peter Lyster

Quantitative assessment of performance: Prologue—this approach of this team is applicable to cases of loops where there are a large number of operations (say, 1,000) on small numbers of variables (say, 10).

a. SAXPY. It is probably not cost effective to build a custom chip to do this because there are not enough operations per variable in the loop, and this is just part of a large computation.

b. Large-stride vector fetch and store. This may require a sorting procedure (on a front end) in order to avoid the data being strided across processors; we did not attempt to characterize this. On-processor operations may be then customized.

c. Irregular gather/scatter. Similar comments as b. The GRAPE methodology may be useful for plasma physics (e.g., fusion). The only problem is the need to sort the particles on the front end, and the need for many more particles in plasma simulations.

d. 3-D Jacobi kernel. The array data may be domain-decomposed in traditional manner (as in message-passing) with appropriate replication of boundaries. This should incur the 'standard' cost of message passing; see back-of-the-envelop calculation at the end of this paragraph. Providing the conditions as described in the prologue above are satisfied, this algorithm may be customized on-chip. In fact, an atmospheric model with fixed resolution (typically 10^6 gridpoints these days, and rising not all that rapidly) and large operational complexity per gridpoint may be amenable to this approach. For the example that was given, the GRAPE approach may be particularly useful if n_c is 'large' as computer complexity grows like n_c^2.

   Attempt (maybe facile) at back-of-the-envelope characterization of Jacobi kernel.

   $n_x = n_y = n_z = 10^3$ implies 10^9 words (gridpoints, cells, ...). This may be excessive for GRAPE which is now aimed at 10^6 particles, and where cost effectiveness partly comes from small memory requirements.

   The inner loops over m and mp have about $nc * nc * 5$ ops. For $nc = 5$ this gives 125 ops/gridpoint, for $n_c = 150$ this gives 10^5 ops/gridpoint. The latter is not dissimilar to the proposed GRAPE machine parameters.

   For the case $n_x = 10^3$ and $n_c = 150$: If the gridpoints are aggregated in standard domain decomposition for message passing, there are 10^5 gridpoints per processor, and there are $6(10^5)^{2/3}$ gridpoints that need to be communicated per 'timestep' per processor. Assuming an interprocessor bandwidth of 10^{11} bits per second, and 64 bits per gridpoint then the communication time is $150 * 64 * 6 * (10^5)^{2/3}/10^{11} = 10^{-3}$ seconds. The time to perform the calculations on chip is approximately $10^5/10^5 = 1$ second. So the ratio of $T_{\mathrm{comm}}/T_{\mathrm{calc}} = 10^{-3}$ and the time per timestep is one second (not too shabby).

This methodology is still fairly restricted, notably there are many flops per word, but it may be useful for future applications that require increased speed at fixed resolution. Climate models with modules that couple gridpoint, such as radiation, would present a difficult problem.

e. Tree-structure. Unclear.

Team 6: HPAM Detailed analysis of question 9

a. SAXPY

The amount of parallelism available in this code is 10^8. This degree exceeds the total number of processors in the HPAM. The problem would be divided among the different levels, such that the total execution time is approximately equal.

b. Large stride vector

The degree of parallelism is 10^8, but the kernel is dominated by accesses to a vector with large stride; no flops. This kernel would be executed on one level, the lowest level, for which the memory latency is less than going between levels.

c. Irregular gather/scatter

The degree of parallelism is 10^8, but the kernel is dominated by irregular, indirect memory accesses. This kernel would be executed on one level, the lowest level. No flops.

d. 3-D Jacobi Kernel

Reorder the loops to change the dependency structure and assign the innermost loops to the processors at the lower level.

e. Tree

Divide the 10^9 node tree into subtrees with 10^4 nodes subtrees and distributed these subtrees among the processors and each processor perform a comparison to see if the original 10^4 tree is a subtree of the partitioned tree.

Performance of Kernels for the HPAM Architecture

The answers to the performance of some of the kernels for the HPAM Petaflop Point Design are given for two different conditions:

Case A: the entire HPAM machine is dedicated to the kernel

Case B: the kernel is a part of a large application for which other parts of the application are being executed simultaneously with the kernel; for this case we identify the best level for the kernel (assuming it will be executed on only one level) and describe the performance.

For all cases, $n = 10^8$ and arrays are presumed appropriately dimensioned.

a. SAXPY

```
do i = 1, n
   c(i) = 3.141592653589793 * a(i) + b(i)
enddo
```

It is assumed that the COTS will have extended double precision representation of floating point numbers, allowing for 128-bit floating point numbers. The value of pi would be represented by a 128-bit number and computations would be executed using this format to maintain the accuracy.

Case A: The amount of parallelism available in this code is 10^8. This degree exceeds the total number of processors in HPAM. The problem would be divided among the different levels, such that the execution time of each level is approximately equal; the load assigned to each level would be inversely proportional to the speed of the processors used in the different levels.

Case B: Because of the high degree of parallelism and the regularity of memory accesses, the lowest level (bottom level) of HPAM is best for this kernel.

b. Large stride vector

```
do i = 1, n
   b(131*i) = a(131313 * i)
enddo
```

Transformed to:

```
do i = 1, n
   tempb(i) = a(131313 * i)
enddo

do i = 1, n
   b(131 * i) = tempb(i)
enddo
```

This problem is memory bound, requiring only memory accesses with a large stride; no floating point operations are used. The memory access pattern is such that there is no temporal or spatial locality. Each reference to the arrays will incur a cache miss. The approach to efficiently executing this problem involves a combination of code transformation, prefetching and post-storing. Given the large stride of the accesses, it would be best to execute the kernel on one of levels that has the required memory. The level of choice would have memory access times and parallelism degree (i.e., number of processors) that yield the best execution time. Execution on only one level avoids the high cost of intralevel communication versus interlevel communication.

Further, the elements of "a" would be prefetched as early as possible; the results would be stored in "b" as late as possible to allow access of "a".

c. Irregular gather/scatter

```
do i = 1, n
   idx(i) = 1 + mod(13131313 * i, n)
enddo

do i = 1, n
   b(idx(i)) = a(i)
```

This problem is very similar to the previous problem, but now the array accesses are indirect and irregular. Again prefetching would be used. The array "idx" would be accessed ahead of the array "b"; the array "b" would be accessed ahead of array "a". The array "a" has a very regular access, exhibiting spatial locality.

Also, given the irregular large stride and irregular access of "b", it would be best to execute the kernel on one of the levels that has the required memory. The level of choice would have memory access times and parallelism degree (i.e., number of processors) that yield the best execution time.

d. 3-D Jacobi Kernel

```
do i = 1, nz
   do j = 1, ny
      do k = 1, nx
         do m = 1, nc
            do mp = 1, nc
               a(i,j,k,m) = u(mp,m) * (b(i+1,j,k,mp) + b(i-1,j,k,mp)
                   + b(i,j+1,k,mp) + b(i,j-1,k,mp) + b(i,j,k+1,mp) +
                   b(i,j,k-1,mp)) + a(i,j,k,m)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

The above kernel represents a jacobi kernel for a 3-D grid problem, where each node corresponds to a supernode with $nc \times nc$ regular nodes. The computation involves a reduction to accumulate the results into the array "a". The reduction takes place for the "m" loop and is accumulated over the "mp" loop.

(i) $n_c = 5$, $n_x = 1,000$, $n_y = 1,000$, $n_z = 1,000$
For this case, since n_c is small, the m and mp loops can be mapped to a single processor. The i, j, and k loops can be mapped to different processors. The combination of these three loops results in a parallelism of 10^9.

Case A: The i, j, and k loops can be distributed among all the processors in the HPAM machine, such that the execution time of each level is approximately equal. The load assigned to a processor would be inversely proportional to the processor speed. Each processor would execute the reduction for a given value of i, j, and k. It is assumed that the processors have the appropriate values of b. The array u is small enough so that it can be replicated in all the processors.

Case B: Given the high degree of parallelism of the three loops and the fact that no interlevel communication is needed for the reduction for a specified i, j, and k, these three loops can be distributed over the lowest level of HPAM.

(ii) $n_c = 150$, $n_x = 100$, $n_y = 100$, $n_z = 100$

Now the core of the m and mp loops would be executed a total of 22,500 times with some dependencies in the reduction. This is three orders of magnitude larger than the previous case. The combination of the i, j, and k loops has a degree of parallelism of 10^6.

Case A: Again the i, j, and k loops can be distributed across the entire machine with one major difference. The m and mp loops can be distributed across a group of processors within the same level. The processors within a group would have the same i, j, and k values, but different values of m and mp. In this way, the reduce operation would occur over a small number of processors within the same level, thereby avoiding excessive interlevel communication.

Case B: The kernel would be executed on the lowest level because of the 10^6 degree of parallelism for the i, j, and k loops. The m and mp loops would be distributed over a group of processors, similar to Case A, but all the processors are now in the lowest level.

e. Processing of tree-structured data

Problem: Is there a subtree of size 10^4 nodes within a similar tree of size 10^9 nodes.

The solution method would depend on the algorithm chosen for this problem. A perhaps naive approach is as follows: First construct all possible distinct (not necessarily disjoint) subtrees with 10^4 nodes of the original tree. Next, test whether each tree matches the desired subtree. Explicitly generating and storing each subtree would be extremely memory intensive, but certainly possible if memory was available. Alternatively, assuming the original tree is decomposed into distinct disjoint subtrees, all subtrees that overlap with one of these subtrees can be generated within a topological neighborhood of a given subtree and processor assigned to it. Distribute these subtrees among the processors (one or more per processor depending on the relative speed of the processors). Let each processor perform a comparison to see if the desired 10^4 subtree is a subtree of the subtree(s) in the processor or those subtrees that result from adding nodes from neighbor processors (this would require an algorithm). This can be distributed across the entire HPAM machine. Interlevel and intralevel communication would be required between processors with adjacent edges of the tree. This communication would hopefully be infrequent relative to the computational requirements of the local tree comparison.

An alternative method is to represent the tree as a two-dimensional sparse matrix, such that $R(i,j) = 1$ if j is a child of i and 0 otherwise. Then, pattern matching for the sparse matrices would be used.

Team 7: I-ACOMA Detailed analysis of question 9

a. SAXPY

Single precision. The operations will be performed within a cluster, assuming the data is in memory (150 cycles to get it). Processors cycle at 1 GHz (1 ns cycle time). Loop is fully parallel, so each of the 10 processors does 1/10th of the operations. For simplicity, assume that the stores are non-blocking, therefore, complete in the background. Assume that the loads are blocking (only one load per processor at a time). Assume that there is no prefetching of data. Assume that, compared to cache misses, the mpy and add operations take negligible time. Cache lines are 64 bytes long.

So, each processor needs to read $4 * 10^8/10$ bytes of a() and $4 * 10^8/10$ bytes of b(). That is $8 * 10^7$. Every 64 bytes, there is a miss. This is then $8 * 10^7/64 = 125 * 10^6$ misses. Each miss costs 150 ns so: $1.25 * 10^6 * 150 * 10^{-9} = 0.18$ seconds.

Note that with prefetching, this number can be reduced considerably.

b. Large-stride vector fetch and store

Similar case as above, except that every single load misses. There are 10^7 loads per processor, each taking 150 ns. The total time is then $10^7 * 150 * 10^{-9} = 1.5$ seconds.

c. Irregular gather/scatter

Takes the same amount of time than a) because we are reading two arrays: a() and idx() sequentially. The store (which stores in b() in a random manner) does not take time in our assumptions. Total time = 0.18 seconds.

d. 3-D Jacobi kernel

First, we reorganize the loop nest as follows:

```
do k
   do m
      do mp
         do j
            do i
```

and we parallelize the k and m loops. There are $n_z * n_c = 5,000$ tasks that can be executed in parallel. We give one to each processor. Each processor then executes the loop nest:

```
do mp
   do j
      do i
```

for a given value of (k, m).

As usual, time is determined by the misses. The misses in array u() are negligible. Let us consider the misses in b() and a().

For a():

A processor reads the plane $a(i, j)$ which is $4 * n_x * n_y = 4 * 10^6$ contiguous bytes. This is reused n_c times thanks to loop `do mp`. Suppose that we block it so that it fits in the primary cache. Then, we only need to bring the data once. As usual, every 64 bytes there is a miss. Each miss is serviced in 150 ns. So, total time is $150*10^{-9}$seconds $*4*10^6/64 =$ 9.4ms.

For b():

A processor reads three planes: $b(i, j, k)$, $b(i, j, k-1)$, and $b(i, j, k+1)$, for a given value of mp. This is $4 * 3 * n_x * n_y = 4 * 3 * 10^6$ contiguous bytes. However, it will read n_c different such three-plane chunks. So, the total data will be $4 * 3 * n_x * ny * nc = 4 * 3 * 5 * 10^6$. We will again block, and every 64 bytes suffer a miss. So, total time is $150 * 10^{-9}$seconds $* 4 * 3 * 5 * 10^6/64 = 141$ms.

So, the total Jacobi for the large problem size without prefetching takes $141 + 9.4$ms $=$ 150.4ms.

For the small problem size, the time is

For a(): $150 * 10^{-9}$seconds $* 4 * 10^4/64 = 0.09$ms

For b(): $150 * 10^{-9}$seconds $* 4 * 3 * 150 * 10^4/64 = 42.3$ms.

Total time $=$ 42.39 ms

e. Processing tree structured data

This is a complex algorithm that can take a variable amount of time. We offer just some intuition here.

we need to compare a tree with 10^4 elements to one with 10^9 elements. Assume that each element is 12 bytes: a leftpointer, the integer, and the right pointer. We can partition the 10^9 tree among the 2,000 nodes (it fits in memory): $10^9/2 * 10^3 = 5 * 10^5$ elements per node. Then, each of the 10 processors in a node works on 1/10 of the elements; that is each processor works on $5 * 10^4$ elements of the large tree.

Each processor will therefore read:

Small tree: $10^4 * 12$ bytes $= 12 * 10^4$ contiguous bytes. One miss every 64 bytes means we have 1,875 misses (negligible).

Fraction of the large tree: $5 * 10^4$ elements. Suppose that the data are not contiguous. We will suffer $5 * 10^4$ misses (one per element) which cost 7.5 ms.

Assume that all the time is taken by the loading of the data; the operations of the comparison may be assumed free. Then, this first pass of comparisons will cost 7.5 ms.

If none of the processors produces a partial match or one of the processors finds the whole tree, the program finishes. If not, we have to have the processors that produced the partial matches work on their neighbor's chunk of the large tree to finalize the match or prove it wrong. That would be another 7.5 ms. However, the data may now be remote, not local in the cluster. Since a remote miss is at least one order of magnitude more expensive, the operation would take 75 ms.

Overall: the total time is then 82.5 ms.

Team 8: NJIT Detailed analysis of question 9

Execution times: (This is only information they gave!)

a. SAXPY loop: 7.08 microseconds

b. Large-stride vector fetch and store: 51.46 microseconds

c. Irregular gather/scatter: 25.76 microseconds

d1. Jacobi kernel 1: 10.17 milliseconds

d2. Jacobi kernel 2: 135.78 milliseconds

2. Programming and Execution Models; Programming Languages

Team 1: MORPH Programming

— Sequential language + messaging or put/get

— Parallel object systems + coordination systems

— Shared memory (various adaptations of consistency protocols)

Team 4: GRAPE Programming

There are driver programs that call specialized functions. The specialized functions are implemented in hardware, while the drivers are written in F77/C/C++. There exit programmable hardwares that handle a range of different force laws.

Team 6: HPAM Programming

Model: All levels appear as shared memory, but each level can be either message passing or shared memory. The programming of each level can be explicit to the user.

Languages: Fortran (as supported by POLARIS), but there could be others.

Team 7: I-ACOMA Programming

This is a pure shared memory with the hierarchy invisible to the programmer. Prefetching could be allowed, making some aspects visible.

Team 8: NJIT Programming

The user is presented a fully-connected system. Execution model: Two-dimensional MIMD generalized hypercube with 10,368 processors (or 12,800 processors for a fault-tolerant system with I/O capabilities). This generalized hypercube is a two-dimensional system where the processors in each of the two dimensions are fully connected.

4. Synchronization mechanisms; hardware fault tolerance facilities

Team 1: MORPH Synchronization

— basic atomicity operations (fetch and increment, compare and swap, additional programmable operations)

— network, processor, and network reconfigurability to isolate faulty nodes and provide a uniform machine image

Team 4: GRAPE Synchronization

The algorithm are self synchronizing through the drive modules. No opinion regarding fault tolerance facilities.

Team 6: HPAM Synchronization

Synchronization: Locks and barriers within levels and between levels. Software based approach and/or backbone approach (including bypasses) in addition to communication backbone.

Fault tolerance: Fall back on different levels; harder to handle within a level.

Team 7: I-ACOMA Synchronization

Support for basic atomic read-modify-write to variables. This support will be used by the libraries to implement locks and barriers, for example.

Team 8: NJIT Synchronization

Up to 2,432 additional processors for fault tolerance (no performance degradation due to "reconfiguration").

5. Mass Storage, I/O and Visualization Facilities

Team 1: MORPH I/O

– Direct connection of peripherals to the scalable interconnects, protection and coordination mechanisms for arbitration of access.

Team 4: GRAPE I/O

There are typically 10^6 particles, the algorithm performs 10^7 operations/particle/timestep, on average there are a total of 10^6 timesteps. Clearly, the memory and I/O to flop ratio is pretty small.

Team 6: HPAM I/O

Mass Storage: Network connection between levels

I/O: Multiple I/O co-processors

Team 7: I-ACOMA I/O

Distributed I/O: One or more disks per node. All I/O is shared.

Team 8: NJIT I/O

All three features are implemented in our system.

6. Support for extended precision (i.e., 128-bit and/or higher precision integer and floating-point arithmetic)—hardware or software.

Team 1: MORPH Precision

This support could be added if necessary.

Team 4: GRAPE Precision

The core operations are 32 bit. Some operations that need higher accuracy are 64 bit. The latter (64 bit) and may need to be extended to 128 bit. Presumably, this is just a matter of changing the specifications to the chip maker.

Team 6: HPAM Precision

— 64 bits assured

— 128 bits at top level

— 128 bits in software at lower levels

Team 8: NJIT Precision

128-bit floating-point arithmetic

7. Which of the 10 key federal grand challenge applications will the proposed system address (aeronautical design, climate modeling, weather forecasting, electromagnetic cross-sections, nuclear stockpile stewardship, astrophysical modeling, drug design, material science, cryptology, molecular biology).

Team 1: MORPH Applications

Federal Grand Challenge Programs (drug design, molecular biology, astrophysical modeling) + interactive 3D visualization

Team 4: GRAPE Applications

Astrophysical modeling. To the extent that plasma physics problems (using particles) may be addressable, then include materials science and nuclear fusion. The question of wider applicability doesn't appear to have been addressed by the team (see also comments in prologue to Section 9 for this team) and this may be a fruitful area to investigate, notably climate modeling, see item 9d for this team.

Team 6: HPAM Applications

Architecture mapping to federal grand challenges was a general purpose machine with target applications for validation: aeronautical design (CFD, FEM), astrophysical, and molecular biology

Team 7: I-ACOMA Applications

Can handle all types of applications easily because it is a general-purpose shared-memory machine. Specially easy to program for irregular applications and applications with fine-grain sharing.

Has hardware support for sparse computations and databases.

Sparse type computations: Because they are difficult to parallelize. Databases: Because they are very important. For both types of applications, we have special hardware support.

Team 8: NJIT Applications

Climate modeling and weather forecasting

8. Reliance on novel hardware and/or software technology, to be developed by the commercial sector.

Team 1: MORPH Custom Technology

No novel requirements of the commercial sector, we will develop technology for hardware synthesis appropriate for customization of cache and network management, as well as hardware performance monitoring.

Team 4: GRAPE Custom Technology

This uses custom chip designs, but rather conservative technology. For applications where this approach is appropriate (i.e., where a lot of operations are performed on not a lot of memory) the chips need to be designed by scientist/engineers, however they don't cost a lot to manufacture, or stress fabrication methodology.

Team 6: HPAM Custom Technology

Hardware: PIM, Optical Interconnects

Software: Identification of different degrees of parallelism in program

Team 7: I-ACOMA Custom Technology

The system only needs to have extra support in the processor chip (to support multithreading, COMA, and the hardware for sparse computations).

It is not too revolutionary. Assumes natural progress in

— faster and more complicated processors

— complicated memory hierarchies

— effectiveness of parallelizing compilers

Team 8: NJIT Custom Technology

Our design could benefit from improvements in current technology. Its success, however, does not really depend on any hardware innovations. Improved compiler technologies, better programming languages, and performance visualization strategies could enhance usability; this is true for any system designed for near Petaflops performance.