

## Using Machine Affinity to Increase Science Throughput (Machine Affinity Characterization of the HPCMP Workload)

Allan Snaveley, Anthony Gamst, Laura Carrington, Mustafa Tikir, and Michael Laurenzano  
*San Diego Supercomputer Center, University of California, San Diego, CA*  
 {allans, lcarring, agamst, lcarring, mtikir, mlaurenzano}@sdsc.edu

### Abstract

*Machine affinity is the observed phenomena that some applications benefit more than others from features of high performance computing (HPC) architectures. When considering a diverse portfolio of HPC machines manufactured by different vendors and of different ages, such as the set of all supercomputers currently operated by the Department of Defense High Performance Computing Modernization Program, it should be obvious that some run a given application faster than others do. Therefore, almost every user would request to run on the fastest machines. But an important insight is that some applications benefit more from the features of the faster machines than others do. If allocations are done in such a way that applications that benefit the most from the features of the fastest machines are assigned to those machines then overall throughput across **all machines** is boosted by more than 10%. We exhibit exemplary empirical analysis and provide a simple algorithm for doing allocations based on machine affinity. The net effect is like adding a new \$10M supercomputer to the portfolio without paying for it.*

### 1. Introduction

Currently, the Department of Defense (DoD) Supercomputing Resource Centers (DSRCs) of the High Performance Computing Modernization Program (HPCMP) operate the supercomputers listed in Table 1. The machines in Table 1 span a broad set of architectural features covering most of the major vendors (Cray, IBM, SGI, Dell, Linux Network), and processor chip makers (Intel, AMD, IBM), as well as most of the major interconnect technologies (SeaStar, Infiniband, Federation, NUMALink) and various file and input/output (I/O) subsystems (Lustre, GPFS). Their combined value factoring in depreciation is about \$80 million<sup>1</sup>. Their purpose is to run scientific applications of strategic

importance to the Department of Defense. One would like to know “which is faster?” and the answer of course depends on the application and input. The following is a set of applications used to evaluate systems during each Technology Insertion (TI) procurement cycle: AMR, AVUS, CTH, GAMES, HYCOM, ICEPIC, and LAMMPS. The applications and test cases (Standard and Large) are described in more detail below. Table 2 shows their relative speeds averaged across several CPU counts on each of the above machines normalized to the fastest time (the fastest time scores 100).

**Table 1. Machines operated by the High Performance Computing Modernization Program**

Architectures Used in Study									
DSRC	Name	Make	Model	Chip Set	Processor Speed (GHz)	Interconnect	Number of Cores	Cores per Node	Operating System
ERDC	Diamond	SGI	Altix ICE	Intel Xeon QC	2.8	DDR-4 InfiniBand	15,360	8	SUSE Linux
ARL	Harold	SGI	Altix ICE	Intel Xeon QC	2.8	DDR-4 InfiniBand	10,752	8	SUSE Linux
MHPCC	Mana	Dell	PowerEdge M610	Intel Xeon QC	2.8	DDR Infiniband	9216	8	Linux
NAVY	Delvinc	IBM	Power6	IBM P6 DC	4.7	Federation	4800	32	AIX
NAVY	Einstein	Cray	XT5	Cray Opteron QC	2.3	SeaStar2+	12736	8	CNL
ERDC	Jade	Cray	XT4	Cray Opteron QC	2.1	SeaStar2	8608	4	CNL
ARL	MIM	LINK	ATC	Intel Xeon DC	3	Infiniband	4400	4	SLES 9
ERDC	Sapphire	Cray	XT3	Cray Opteron QC	2.6	SeaStar	8192	2	CNL
NAVY	Babbage	IBM	Power5+	IBM p575 DC	1.9	Federation	2912	16	AIX
AFRL	Hawk	SGI	Altix 4700	Intel Titanium DC	1.6	NUMALink4	9216	512	SLES 10 - Pro Pack 5

If we pick just one of these test cases and a CPU count then we can answer the question “which machine is fastest” definitively for that application test case; for example if we run AMR on the “Standard” benchmark input at 128 processors, then the machines are ranked (fastest to slowest) as shown on the left in Table 3 below. But if we choose a different test case, CTH Standard 128, we get the ranking on the right. (Sapphire has two distinct configurations).

<sup>1</sup> Larry Davis in personal correspondence

**Table 3. Ranking machine by performance on AMR STD 128 (on the left) versus CTH STD 128 (on the right)**

AMR standard with 128 Tasks		CTH standard with 128 Tasks	
1. MHPCC Mana		1. ERDC Diamond	
2. ERDC Diamond		2. ARL Harold	
3. ARL Harold		3. MHPCC Mana	
4. NAVY Davinci		4. NAVY Davinci	
5. NAVY Einstein		5. NAVY Einstein	
6. AFRL Hawk		6. ERDC Jade	
7. ERDC Jade		7. ARL MJM	
8. ERDC Sapphire		8. ERDC Sapphire	
9. ARL MJM		9. NAVY Babbage	
10. NAVY Babbage		10. ARL Hawk	

Since Mana runs AMR Standard 128 more than 2x faster than Diamond, it can be seen there is quite a range of capabilities across all platforms. And although there is not one machine that is fastest on all applications, there are clearly faster and slower machines as a trend (the slower machines tend to be the older machines), and most users would like to run on the faster machines.

There is a formal allocation process for DoD Challenge Projects to determine who gets to run where: every year users of the systems get an allocation of hours on (a subset of) these machines based on reviews of proposals they submit to the Program describing their science and computational needs. Strategic Service needs (Army, Navy, and Air Force) are also factored in, and each application gets a score, based on technical merit and strategic importance, that determines if they get a full allocation on the machines they prefer. Typically, users request a certain number of CPU hours on particular machines and may also express 2<sup>nd</sup> and 3<sup>rd</sup> choices. Many users request the fastest machines for their applications, while some request machines they are familiar with. Requests are granted (and in some cases cut or moved to slower machines) based mainly on technical merit and Service needs. If two users both request the fastest machine for their work but there is not sufficient capacity to grant both in full, a good question becomes “who deserves it more?” In most cases this decision is made by the allocations committee, based on Service needs and ranking of a proposal’s technical merit. But we wish to suggest that perhaps the tie should be broken by “whoever benefits from it more by running faster as a percentage relative to their next choice”. For example, let’s assume proposal A is deemed the better proposal by Service needs and technical merit, but only benefits by running 1% faster on the fastest machine versus the 2<sup>nd</sup> fastest; while proposal B, also ranked highly, would run 20% faster on the fastest machine versus the 2<sup>nd</sup> fastest. Putting the higher-ranked proposal on the second fastest machine would probably not even be noticeable to its

users, while proposal B would in effect get 20% more computational work done for the same CPU hours (or, by another way of looking at it, get in effect a 20% greater allocation, or by another way of looking at it, get in effect a 20% bigger machine to run on). Since the high cost (dollar) value of the entire portfolio is based just on its ability to run large volumes of strategic applications fast, strategies that can boost efficiency, such as this, ought to be carefully considered.

## 2. A Simple but Realistic Example

Let us make this line of reasoning a bit more concrete. We carry out a simplified allocation using 12 applications and 12 machines. We oversimplify for pedagogical purposes (we will restore the full-complexity in the next section). Let us assume the 12 jobs are all 128 CPU jobs that would run for one hour on Babbage, and the capacity of the 12 machines are also 128 CPUs (so we consider a small subset of the actual machines in this allocation exercise), and we wish to assign these jobs to these subset machines in a way that increases computational throughput.

Here (Table 4) are the observed runtimes of the 12 application codes and inputs run on 128 processors of the 12 machines where now the runtimes are expressed as normalized speedup fraction over Babbage i.e., [(runtime on Babbage)-(runtime)]/(runtime on Babbage). This way, the fraction for Babbage is always 0. Table 3 is then a 12x12 matrix, with the rows labeled by machines, and the columns labeled by application/inputs, and each entry is a normalized speedup of running that application/input pair on the machine that labels the row over Babbage (a negative number means it runs slower than Babbage). Note the rank order for 128 CPUs specifically is not the same rank order as that for all CPU counts averaged (i.e., the rank order of Table 4 is not the same rank order as Table 2.)

**Table 2. Relative speeds of the machines on different applications and inputs (na means the application is not run on that machine). Speeds are normalized to 100 where 100 is the speed on the fastest machine.**

Matrix Summary of all Application Runtimes													
Architecture	AMR Standard	AMR Large	AMR Large	CTH Standard	CTH Large	GAMES Standard	GAMES Large	HYCOM Standard	HYCOM Large	ICEPIC Standard	ICEPIC Large	LAHHS Standard	LAHHS Large
ERDC Diamond	83	89	100	100	89	87	54	37	85	100	100	100	100
ARL Harold	78	100	99	93	96	84	33	88	89	92	93	86	80
MHPCC Mana	100	97	82	74	89	60	38	86	90	93	85	86	75
NAVY Davinci	75	72	79	84	100	56	31	100	100	44	53	91	98
NAVY Einstein	55	68	48	48	70	100	100	40	41	40	44	48	47
ERDC Jade	48	59	44	38	64	52	54	35	36	37	40	46	44
ARL MJM	43	49	42	39	63	48	na	40	46	66	55	69	56
ERDC Sapphire	44	56	46	41	66	45	28	35	31	27	42	52	50
NAVY Babbage	37	40	41	49	47	51	na	31	23	23	22	40	36
AFRL Hawk	49	40	33	26	43	21	na	40	36	21	na	23	30



**Table 4. Speedups relative to Babbage of machines on 128 CPU application/inputs**

Name	AMR S	AVS L	CTH S	GMS S	HYC S	HYC P	ICE S	ICEPIC	HYC P	HYC S	AVS L	ICE S
Mjm	0.19	0.02	-0.20	-0.12	0.12	0.66	0.39	0.34	0.66	0.12	0.02	0.34
Jaws	0.26	0.04	-0.30	-0.12	0.06	0.63	0.40	0.36	0.63	0.06	0.04	0.36
Babbage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hawk	0.31	-0.18	-0.77	-0.94	-0.17	-0.14	-0.79	-0.25	-0.14	-0.17	-0.18	-0.25
Sapphire_CAT	-0.17	0.01	-0.24	-0.37	-0.21	0.41	-0.69	-0.21	0.41	-0.21	0.01	-0.21
Sapphire_CNL	0.09	0.09	-0.26	-0.15	-0.23	0.38	0.19	0.26	0.38	-0.23	0.09	0.26
Jade	0.16	0.06	-0.32	0.42	-0.18	0.40	0.09	0.15	0.40	-0.18	0.06	0.15
Einstein	0.29	0.11	-0.01	0.47	0.01	0.48	0.18	0.26	0.48	0.01	0.11	0.26
DaVinci	0.50	0.63	0.44	0.09	0.61	0.50	0.58	0.62	0.50	0.61	0.63	0.62
Harold	0.66	0.59	0.51	0.41	0.54	0.78	0.54	0.54	0.78	0.54	0.59	0.54
Mana	0.64	0.59	0.48	0.11	0.52	0.77	0.54	0.55	0.77	0.52	0.59	0.55
Diamond	0.67	0.59	0.50	0.43	0.53	0.78	0.61	0.63	0.78	0.53	0.59	0.63

In this toy example, it should be clear we have to assign one application to each machine (we can't increase throughput by having a machine sit idle), and it should be clear we cannot assign more than 1 job to a machine (since we restrict ourselves to just 128 processors and consider that "the machine" and the jobs are 128 processors each). So we have to allocate *exactly* one application to each machine. So how shall we proceed? The usual allocations method, as mentioned above, assigns jobs to machines without any regard to their relative speedups; rather other external criteria such as technical merit is used to make the allocation decision. So to emulate the effect of allocating applications to machines without regard to speedup let's just pick any allocation. For simplicity let us assign application *i* to the machine on row *i* where *i* is the column of the application in the table above. To be precise, let us multiply the table above by a diagonalized allocation matrix shown below.

**Table 5. A random allocation matrix and the resulting speedup of the applications**

Mjm	1	0	0	0	0	0	0	0	0	0	0	0	
Jaws	0	1	0	0	0	0	0	0	0	0	0	0	
Babbage	0	0	1	0	0	0	0	0	0	0	0	0	
Hawk	0	0	0	1	0	0	0	0	0	0	0	0	
Sapphire_CAT	0	0	0	0	1	0	0	0	0	0	0	0	
Sapphire_CNL	0	0	0	0	0	1	0	0	0	0	0	0	
Jade	0	0	0	0	0	0	1	0	0	0	0	0	
Einstein	0	0	0	0	0	0	0	1	0	0	0	0	
DaVinci	0	0	0	0	0	0	0	0	1	0	0	0	
Harold	0	0	0	0	0	0	0	0	0	1	0	0	
Mana	0	0	0	0	0	0	0	0	0	0	1	0	
Diamond	0	0	0	0	0	0	0	0	0	0	0	1	
AVG	0.19	0.04	0.00	-0.94	-0.21	0.38	0.09	0.26	0.50	0.54	0.59	0.63	0.17

The result of multiplying each row to assign each application to exactly one machine is shown in the last row of Table 5, i.e., now the resulting speedup of that application over Babbage. Since the order of machines and applications in the table are arbitrary this is in effect a random allocation policy.

About any such assignment there is an interesting figure of merit shown above in the last row. The last column lists the average of all and it is simply the average speedup of the entire workload over just running it on Babbage (i.e., the average value of the entry selected by the machine assignment matrix). Table 5 shows that in this arbitrary assignment of jobs to machines, we are running the entire workload 17% faster than if all the machines were Babbage. That makes sense, because in fact, a majority of the machines are faster than Babbage, so on average you should see some speedup relative to it.

But the question remains "Can we do better"? And in fact it is easy to do better; a few simple ideas yield a

much better assignment. The highest speedup entry in Table 4 for is HYCOM P on Harold (column 9 row 10 at a value of 0.78), so let us try to obtain that by just interchanging column 9 and column 8 machine assignments. And the single worst "speedup" in Table 3 is 0.94. Unfortunately, we got that in our naive assignment by running (GAMMES S on Hawk) so let us just try to avoid that and trade jobs with Mjm (have Hawk run AMR and Mjm run GAMMES). Just these two moves to obtain the best and eliminate the worst possibilities yields a new job assignment; this job assignment matrix is shown below and is 28% faster than if all the machines were Babbage. And since we went from 17% faster than Babbage to 28% faster, it is almost as though we added capability of >10% to the machine portfolio, just by making a modest effort to improve job assignments. (Table 6)

**Table 6. A job allocation matrix that is 11% better than the random one**

Mjm	0	0	0	0	1	0	0	0	0	0	0	0	
Jaws	0	1	0	0	0	0	0	0	0	0	0	0	
Babbage	0	0	1	0	0	0	0	0	0	0	0	0	
Hawk	1	0	0	0	0	0	0	0	0	0	0	0	
Sapphire_CAT	0	0	0	0	0	1	0	0	0	0	0	0	
Sapphire_CNL	0	0	0	0	0	0	1	0	0	0	0	0	
Jade	0	0	0	0	0	0	0	1	0	0	0	0	
Einstein	0	0	0	0	0	0	0	0	0	1	0	0	
DaVinci	0	0	0	0	0	0	0	0	0	0	1	0	
Harold	0	0	0	0	0	0	0	0	0	0	0	1	
Mana	0	0	0	0	0	0	0	0	0	0	0	0	
Diamond	0	0	0	0	0	0	0	0	0	0	0	0	
AVG	0.31	0.04	0.00	-0.12	-0.21	0.38	0.09	0.62	0.48	0.54	0.59	0.63	0.28

### 3. Formalism

Unfortunately, the general problem of assigning jobs to machines to maximize throughput is NP complete<sup>[1]</sup>; that means that in all likelihood we would have to try all possible job assignments (an exponential number of them) to find the best. However, as we just saw in the simple example above, we don't have to find the best one to still boost throughput significantly. We can easily simply try a few and see which of those is better without having to prove it is *best* in any formal sense. Nevertheless, it is nice to have a prescribed method, not just stare at an allocation matrix and try to make local moves by hand. Here is a simple, greedy heuristic that seems to work well in practice. In the following, we assume the more realistic case that the number of hours requested per-year exceeds the number of processor-hours available per-year (so someone has to get cut).

#### A "Greedy" Affinity-based Allocations Heuristic

1. Fill in the m-by-p matrix of speedups and find the biggest entry.
2. Allocate this application to that machine.
3. Subtract the corresponding predicted runtime from the allocatable hours for that machine.
4. Delete that application/column from the matrix, producing an m-by-(p-1) matrix of speedups.

- If a machine is fully allocated, delete that machine/row from the matrix, producing an (m-1)-by-(p-1) matrix of speedups.
- Repeat until all applications have been allocated or the matrix is null.

Now this is an algorithm that can be applied to any allocations *if we know the runtimes of the applications on the machines* (more about this later) and applying it to our toy problem assuming (for simplicity) that each application runs all-year yields:

**Table 7. A job allocation matrix that is 11% better than the random one**

Mjm	0	0	0	0	0	0	0	0	0	0	1	0	0
Jaws	0	0	0	0	0	0	0	0	0	0	0	0	1
Babbage	0	0	1	0	0	0	0	0	0	0	0	0	0
Hawk	0	0	0	0	0	0	0	0	0	0	0	1	0
Sapphire_CAT	0	0	0	0	1	0	0	0	0	0	0	0	0
Sapphire_CNL	0	0	0	0	0	0	0	1	0	0	0	0	0
Jade	0	0	0	0	0	0	1	0	0	0	0	0	0
Einstein	0	0	0	1	0	0	0	0	0	0	0	0	0
DaVinci	0	1	0	0	0	0	0	0	0	0	0	0	0
Harold	0	0	0	0	0	0	0	0	1	0	0	0	0
Mana	1	0	0	0	0	0	0	0	0	0	0	0	0
Diamond	0	0	0	0	0	1	0	0	0	0	0	0	0
	0.64	0.63	0.00	0.47	-0.21	0.78	0.09	0.26	0.78	0.12	-0.18	0.36	0.31

This is now as if we added 14% compute power to our machine portfolio just by doing a better job of job allocations by a simple heuristic. And since the portfolio is worth (conservatively) \$80 million, it is like we added >\$10 million worth of computer hardware to the portfolio without spending any money.

#### 4. Would This Work in Practice?

Obviously if we knew all the runtimes of all applications in advance we could do the heuristic at allocations time, and apparently increase throughput (more about a thorough study below). But how much do we really know about the runtimes of applications that apply to the allocations process? Quite a bit actually; every once in a while someone proposes an entirely new application that has never been run on any DoD machine or any similar machine. In fact, users requesting allocations are required to report runtimes and scalability studies of their applications on any previous machines they have run estimate performance and scalability at least on the machines they are requesting. Very commonly, users are requesting time for running well-known codes such as AVUS or CTH that have been run, at least in some form, on all the existing machines.

It would clearly be possible to run the allocations heuristic with a partially-completed speedup table and/or a speedup table, some of whose entries are estimated. In the case of entirely missing entries one can run the heuristic until there are only missing entries in the table and then make random allocations.

The HPCMP also has a robust performance modeling capability. The framework is capable of predicting performance of characterized applications on the existing

portfolio and future machines with 90% accuracy<sup>[2]</sup>. Some of the entries in the matrix could be performance predictions rather than observed speedups.

#### 5. Future Work: Results of a More Realistic Simulation

We want to know what would happen if we scheduled a larger and more diverse workload in a realistic scenario using real capacities of the machines and employing backfilling. We also want to know the impact of uncertainty in the speedup entries.

In previous work<sup>[3]</sup>, we developed a realistic job scheduling simulator that can consume job log traces and schedule a workload across a set of machines employing state-of-the-art algorithms, such as backfilling, user-utility scheduling, and the like. We modified it to employ the heuristic in making job assignments, and are running it on a synthetic workload derived from the observed runtimes of the following 15 applications and test cases (amr\_std, amr\_lrg, avus\_lrg, cth\_std, cth\_lrg, gamess\_std, gamess\_lrg, hycom\_serio\_std, hycom\_serio\_lrg, hycom\_std (pario), hycom\_lrg(pario), icepic\_std, icepic\_lrg, lammmps\_std, lammmps\_lrg) at processor counts ranging from 48 to 2,048 and some in between (not all jobs come in all CPU counts), a total of 75 different flavors of application/input/CPU counts and totaling a request of greater than 876,000,000 hours (a little greater than the entire capacity of the portfolio). The synthetic workload then resembles the real HPCMP workload in many respects, and early results indicate the throughput boost may be even greater than the simple example seems to suggest.

#### 6. More Details on the Applications and Testcases

**AMR** is a CFD (Computational Fluid Dynamics) code that solves hyperbolic equations on adaptive, structured grids. Its parallelization is via Message Passing Interface (MPI). It is a mixture of about 80% C++ and 20% FORTRAN codes and runs stand-alone code except for MPI libraries. The **AMR Standard** test case simulates a gas-dynamic shock contacting a helium bubble. This is an important problem for combustion research. There are 2 levels of refinement allowed, it runs for 80 time-steps representing about 0.4 seconds of simulation time, and the problem size is very similar to production work loads. A real production problem would run for thousands of time-steps. The **AMR Large** test case simulates a gas-dynamic shock contacting a helium bubble. This is an important problem for combustion research. Four levels of refinement are allowed and it

takes 40 time-steps representing about half a second of simulation. The problem size is very similar to production work loads. A real production problem would run for thousands of time-steps.

**AVUS** is a CFD code, formerly COBALT\_60 that simulates three-dimensional (3D) turbulent viscous flow over irregular geometries. It is grid-based, reads a large grid file and is about 29K lines of FORTRAN 90 code. It uses ParMETIS: 12K lines of C code and its parallelism via MPI, no OpenMP. It runs on Cray XT, IBM Power, SGI Altix, and Linux clusters. The **AVUS Large** test case aka Waverider, models a supersonic or hypersonic vehicle “riding” a shock wave that forms below the vehicle. It uses 31M cells and runs for 400 time-steps.

**CTH** is a CSM (Computational Structural Mechanics) shock-physics code that performs a two-step, 2<sup>nd</sup>-order-accurate Eulerian algorithm used to solve the mass, momentum, and energy conservation equations. It employs an explicit approach that does not require solving a linear system, and has both static and adaptive mesh capabilities. Its parallelism is via MPI and it comprises about 900K LOC, 58% FORTRAN and 42% C. It uses NetCDF, supplied with distribution. The **CTH Standard** test case simulates a rod made of ten materials impacting a plate made of eight materials at an oblique angle of 73.50 with rod length=7.67 cm, rod diameter=0.767 cm rod velocity=1210 m/s rod subdivided through length into ten materials, where plate thickness=0.64 cm, plate velocity=217 m/s plate subdivided through thickness into 8 materials. It uses AMR (Adaptive Mesh Refinement) and a maximum of six levels-of-refinement, 350 time-steps and uses 1GB memory per-process, regardless of the number of processes used. The **CTH Large** test case uses the same model as the Standard case but using a static grid and domain decomposition; the memory requirement per process decreases with increasing processor count and it runs for 300 time-steps. The Standard and Large test cases write large (~10MB) restart files for each MPI process at the start of the execution and again at the end.

**GAMMES** is a CCM (Computational Chemistry, Biology and Materials Science) *Ab Initio* Quantum chemistry code that computes many energy integrals with molecular data in the form of atom positions and electron orbitals. Communication depends on a platform with LAPI, Sockets, SHMEM, and MPI all available. It is 99% FORTRAN, 1% C. The **GAMMES Standard** test case performs a DFT computation to compute the nuclear gradient vector of a “POSS” molecule, using restricted Hartree-Fock calculation with self-consistent field wave functions. The test case is similar to actual workload, and memory usage is about 800 Mbytes. The **GAMMES Large** test case performs an MP2 computation that finds the nuclear gradient vector of a “BC4” molecule using restricted Hartree-Fock calculation with self-consistent

field wave functions. It represents a very machine-stressful workload and has 1.8 GBytes memory usage.

**HYCOM** is a CWO (Climate/Weather/Ocean) code that solves a primitive equation ocean general circulation model. Its communication is MPI (MPI-2 is available), and it is 100% FORTRAN. The **HYCOM Standard** test case is, in essence, a production run. It represents a 26-layer 1/4 degree global model that simulates 1 day at 2,160 time-steps and requires about 0.75 GBytes of memory per-processor, and about 4 GBytes of globally-accessible scratch disk. **HYCOM Large** is an even higher-resolution, larger, more stressful test case with a 26-layer 1/4 degree global model that simulates 3/4-day in 864 time-steps and requires about 0.9 GBytes of memory per processor, and about 23 GBytes of globally-accessible scratch disk.

**ICEPIC** is a CEA (Computational Electromagnetics and Acoustics) code with particle-in-cell plasma physics simulation capability. Ions and electrons move under influence of an electromagnetic field. The fields are calculated on a structured, static grid according to Maxwell’s Equations. It can simulate plasmas contained in complex geometries used in electromagnetic device design. The current version 4,016 has 334K LOC, 100% C++, C. The **ICEPIC Standard** Case carries out a simulation of a magnetron for a high-power microwave source during startup. It features many transients with particles representing electrons being created and moving in a grid-free way throughout the domain, and employs 7267 time-steps to simulate 10-nanoseconds. The **ICEPIC Large** Case is a simulation of a gyrotron source of the air-born version of the active denial system (ADS) with 3,000 time-steps to simulate roughly 355 picoseconds.

**LAMMPS** is a CCM (Computational Chemistry, Biology & Materials Science) classical molecular dynamics code that models particles in a liquid, solid, or gaseous state. It calculates atomic velocities, positions, system energy, and temperature. After equilibration surface tension, radial pressure and phase-change are updated. Post-processing includes pair-correlation function and diffusion coefficients. All actions occur within box (usually orthogonal). It employs distributed-memory message-passing parallelism. It is highly-portable C++ and needs libraries for MPI and single-processor FFT. The **LAMMPS Standard** test case is adapted from the LAMMPS Rhodopsin benchmark model, and simulates an all-atom Rhodopsin protein in a solvated lipid bi-layer. The input file contains molecular topology, force parameters, and initial positions for 32K atoms. The computational domain is being replicated 7 times in each dimension. After replication there are 10.9M atoms, 9.5M bonds, 13.8M angles, 19.5M dihedrals, and 361K improper in an orthogonal box. The calculation carries on for 1,500 time-steps, each

representing 2 femtoseconds. The **LAMMPS Large** test case, adapted from the LAMMPS Embedded Atom Model (EAM) benchmark, is simulating a copper metallic solid. It reads an input file that contains specifications for the inter-atomic potentials to create 108M atoms within an orthogonal box. It employs 2,500 time-steps, with each representing 0.005 femtoseconds.

## Acknowledgements

This work was sponsored in part by the Department of Defense High Performance Computing Modernization Program. The authors wish to thank Mark Cowan for providing much of the raw data and machine and test case descriptions.

## References

1. Garey, M R., D.S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete Problems." *Proceedings of the Sixth Annual ACM Symposium on theory of Computing*, Seattle, WA, April 30–May 02, 1974. STOC '74. ACM, New York, NY, pp. 47–63, 1974, DOI= <http://doi.acm.org/10.1145/800119.803884>.
2. Snavey, A., L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Performance Modeling and Prediction." *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, MD, Conference on High Performance Networking and Computing, IEEE Computer Society Press, Los Alamitos, CA, pp. 1–17, 2002.
3. Lee, C.B. and A.E. Snavey, "Precise and Realistic Utility Functions for User-centric Performance Analysis of Schedulers." *Proceedings of the 16th international Symposium on High Performance Distributed Computing*, Monterey, CA, June 25–29, 2007. HPDC '07. ACM, New York, NY, pp. 107–116, DOI= <http://doi.acm.org/10.1145/1272366.1272381>.