Cosmic Microwave Background Map-Making at the Petascale and Beyond

Rajesh Sudarsan Julian Borrill Christopher Cantalupo
Theodore Kisner Kamesh Madduri Leonid Oliker Horst Simon Yili Zheng
Computational Research Division, Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{rsudarsan, jdborrill, cmcantalupo, tkisner, kmadduri, loliker, hdsimon, yzheng}@lbl.gov

ABSTRACT

The analysis of Cosmic Microwave Background (CMB) observations is a long-standing computational challenge, driven by the exponential growth in the size of the data sets being gathered. Since this growth is projected to continue for at least the next decade, it will be critical to extend the analysis algorithms and their implementations to peta-scale high performance computing (HPC) systems and beyond. The most computationally intensive part of the analysis is generating and reducing Monte Carlo realizations of an experiment's data. In this work we take the current state-of-the-art simulation and mapping software and investigate its performance when pushed to tens of thousands of cores on a range of leading HPC systems, in particular focusing on the communication bottleneck that emerges at high concurrencies. We present a new communication strategy that removes this bottleneck, allowing for CMB analyses of unprecedented scale and hence fidelity. Experimental results show a communication speedup of up to 116× using our alternative strategy.

1. INTRODUCTION

The CMB is the remnant radiation from the Big Bang itself. Last scattered when the Universe first cooled enough for neutral hydrogen to form, some 400,000 years after the Big Bang, it provides the earliest possible image of the Universe. Tiny fluctuations in the CMB across the sky encode not only the basic parameters of cosmology, but also, using the Big Bang as the ultimate particle accelerator, insights into fundamental physics at energies some 12 orders of magnitude higher than those of the Large Hadron Collider beams [9]. However the faintness of these fluctuations requires us to gather and process enormous data sets to achieve sufficient signal-to-noise to decode them, and as a result, CMB data analysis is an extremely computationally intensive endeavor. Since CMB data sets have been growing exponentially in the last two decades, and are projected to continue to do so for at least the next decade, their analysis presents a long-standing challenge to scale the algorithms and implementations to the largest supercomputers available at any epoch.

The main focus of this work is to scale MADmap – a state-of-the-art massively parallel CMB analysis code – to the next generation of peta-scale supercomputers in order to be able to apply it to CMB experiments currently being fielded. Previous work by Cantalupo et.al [6] has shown that the inter-node communication phase in MADmap becomes a bottleneck when it is scaled to more than $O(10^4)$ MPI tasks. The highly irregular communication required in MADmap cannot be efficiently captured via nearest neighbor-based or tree-based all-to-all communication schemes. We present a novel approach that minimizes the data volume communicated, and is generally applicable to any sparse, irregular data distribution.

This paper presents two new optimizations towards alleviating the communication bottleneck and scaling up MADmap. The first one involves implementing the new communication algorithm that replaces MPI_Allreduce with MPI_Allgather operations (discussed in more detail in Section 3.3). We also employ "hybrid" programming to reduce the number of MPI tasks. We utilize one MPI task per socket, together with as many with OpenMP [19] threads as there are cores on the socket. We evaluate the costs and benefits of these optimizations by running a strong scaling experiment on up to 16K cores on four different systems, and show that these combined optimizations significantly reduce the communication bottleneck, achieving a speedup of up to 116×. The goal of this work is to provide an application-level portable optimization for a broad variety of supercomputers used for production CMB analysis.

The remainder of the paper is laid out as follows: Section 2 gives an overview of CMB science in cosmology and a brief introduction to the analysis of current and future CMB experiments; Section 3 details the design and implementation of MADmap, its current communication bottleneck, and the optimizations implemented to alleviate it; Section 4 discusses the experimental setup used to evaluate the performance of these new approaches; Section 5 shows the benefits of these optimizations across a number of HPC systems; finally Section 6 presents our conclusions and directions for future research.

2. CMB SCIENCE

Observations of the CMB have already had a profound effect on our understanding of the Universe. Its very existence sounded the death-knell for the Steady State cosmology, while its extraordinary isotropy posed questions that were effectively addressed by the theory of Inflation, which posits a period of exponential expansion in the first moments after the Big Bang. However, it is the tiny fluctuations in the CMB temperature and polarization that carry the most detailed imprint of our cosmology. Already they have provided the strongest evidence for Inflation, as well as constraining the age, composition and overall geometry of the Universe [16].

At the turn of the millennium, CMB results coupled with

the accelerating expansion of the Universe deduced from observations of type 1a supernovae, led to the surprising but now widely accepted "Concordance Cosmology", in which the Universe is believed to comprise around 70% dark energy, 25% dark matter and 5% ordinary matter [10]. Understanding the nature and origin of these mysterious dark components – some 95% of the mass-energy of the Universe – is now the grand challenge in cosmology and fundamental physics.

Future observations of the CMB promise to yield even greater insight into the foundations of the Universe. The Planck satellite [22] – launched in May 2009, and following in the footsteps of the very successful COBE [27, 8] and WMAP [32] missions – will provide the definitive measurement of the temperature anisotropies, as well as the most detailed polarization observations to date. These results will be an essential complement to the numerous experiments currently being developed to improve our understanding of the dark energy (indeed the expected Planck results are routinely assumed as given in such experiments' performance projections). Beyond Planck, first a series of sub-orbital experiments and ultimately another satellite mission will search for the faintest CMB signal, its B-mode polarization, which is expected to carry, amongst other signals, the imprint of gravity waves emitted during Inflation. Precise measurement of this polarization signal on all angular scales constitutes the next great frontier for CMB research.

2.1 CMB Data Analysis

Most CMB experiments gather their data by scanning the sky with an array of detectors at multiple frequencies to produce a time-ordered data set comprising sky signal (both CMB and astrophysical foregrounds) and instrument noise. These data are reduced to pixelized maps of the intensity (I) and two polarization components (Q, U) of the microwave sky at each observing frequency. These maps are then combined to separate the CMB from the foreground contaminants, and the CMB IQU map-triplet is then used to determine the auto- and cross-angular power spectra of the CMB temperature (T) and E- and B- polarization modes, from which fundamental cosmological parameters can be determined.

This analysis is essentially data compression, progressively reducing the dimensionality of the data from time samples through multi- to single-frequency sky pixels to angular power spectral coefficients and ultimately cosmological parameters. However, the strength of the assumptions required by each step progressively increases, with the first two steps depending only on the experimental data (including performance monitoring data), the next two steps on the statistical and/or astrophysical properties of the sky signal, and the final step on a choice of a cosmology and its parameterization; for this reason the core data products of any CMB experiment are its maps and power spectra.

Under minimal assumptions it is possible to write down maximum likelihood expressions for the maps given the time-ordered data and for the power spectra given the maps, and to show that these represent lossless compression [5]. However, a key feature of CMB data is that their three major components – the CMB itself, foregrounds, and detector noise – are individually correlated, and moreover, each is most simply described in a different domain. Detector noise is piecewise correlated in the time-domain; foregrounds signals are spatially correlated pixel-templates; and the azimuthally symmetric CMB signal can be represented by its angular correlations – indeed it is precisely the strength of these that we want to determine. These various correlations precludes the kind of divide-and-conquer embarrassingly parallel approaches used to analyze very large data sets in disciplines like accelerator physics. Instead, any CMB analysis has to be able to manipulate an entire data set simultaneously and coherently, ideally keeping track not just of the data in each basis, but also of their correlations as they are reduced.

In practice the computational tractability of any CMB analysis depends on two data parameters: the numbers of time-samples ($\mathcal{N}_t$) and sky-pixels ($\mathcal{N}_p$). The first is the product of the number of time streams, their sampling rate(s) and the duration of the observation, while the second comes from the ratio of the fraction of the sky observed to the size of the detector beams. Together these set the overall sensitivity of the experiment and the lower and upper limits of its angular power spectral range.

Using preconditioned conjugate gradient techniques [11] and exploiting the piecewise stationarity of the time-time noise correlations, the floating point operation count for maximum likelihood map-making scales as $\mathcal{O}(\mathcal{N}_t)$. However the maximum likelihood power spectrum estimation flop-count scales as $\mathcal{O}\left(\mathcal{N}_p^3\right)$, making it impractical for data sets with more than a few hundred thousand pixels; while it can still play a role for low-resolution (low multipole) analyses of current and future data sets, it has largely been replaced by Monte Carlo pseudo-spectral methods [12]. These reduce the map-domain operation count by simply taking spherical harmonic transforms of the actual maps to derive so-called pseudo-spectra which are necessarily biased by the maps' incomplete sky coverage (since even full sky observations have to excise the galactic plane) and inhomogeneous, correlated noise. These effects are then corrected for by Monte Carlo methods, simulating the time-ordered data that would have been gathered by the experiment for a set of known sky and noise realizations, mapping these, and using the pseudo-spectra derived from these maps to derive the transfer function needed to convert pseudo to real spectra. These methods therefore scale with the simulation and map-making costs, both $\mathcal{O}(\mathcal{N}_t)$, and both with significant pre-factors.

The dominant computational costs for CMB future missions will therefore come from manipulations of their time-ordered data – primarily simulation (both for mission design and data analysis) and map-making, both often involving repetition over Monte Carlo realizations and/or iterations. Table 2 shows the numbers of time samples gathered or projected for all satellite and a sample of sub-orbital CMB missions. Their evolution has been driven by the demands of CMB science, first for smaller angular scale temperature and then for intrinsically fainter polarization signals, each goal requiring larger data sets to achieve the necessary signal-to-noise. In particular, measuring the B-mode polarization spectrum will requires experiments with 1-2 orders of magnitude more detectors, such as the EBEX [20] balloon-borne, PolarBeaR [23] and QUIET [24] ground-based, and ultimately CMBpol [7, 4] satellite missions.

Over the next 15 years we can expect CMB time-ordered data volumes to grow by three orders of magnitude; coincidentally this exactly matches the projected growth in

Table 1: The evolution of all satellite and selected sub-orbital CMB experiments' actual or projected sample counts over time, driven by the sensitivities required to detect signals both at higher resolution and of lower intrinsic brightness. Details of the proposed next-generation CMBpol satellite are from the EPIC mission concept study.

Date	Experiment	Description	Duration	Streams	Rate (Hz)	Samples
1990-93	COBE	Low-res, T	4 years	6	1	8×10^8
1998	BOOMERanG	Mid-res, T	10 days	16	63	9×10^8
2001-10	WMAP	Mid-res, TE	9 years	40	7.8-19.5	2×10^{11}
2009-11	Planck	High-res, TE	2 years	74	32.5-172	6×10^{11}
2011	EBEX	High-res, TEB	14 days	1406	400	7×10^{11}
2011-13	PolarBeaR	High-res, TEB	2 years	2548	200	3×10^{13}
2012-15	QUIET-II	High-res, TEB	3 years	11200	100	1×10^{14}
2020+	CMBpol/EPIC	High-res, TEB	4 years	1620	1000	2×10^{14}

computing power over the same period assuming a continuation of Moore's Law. Since today's CMB data analyses are already pushing the limits of current HPC systems, this implies that the algorithms and their implementations will have to continue scaling on the leading edge of HPC technology for the next 10 epochs of Moore's Law if we are to be able fully to support first the design and deployment of these missions and then the scientific exploitation of the data sets they gather.

3. MADMAP

3.1 The Current MADmap Package

The last decade has seen the development of a general purpose massively parallel CMB map-making code, MADmap, together with its application to real and simulated data from a number of experiments on many generations of HPC systems. As shown in Figure 1, MADmap has already successfully been scaled through a 100-fold increase in concurrency, 600-fold increase in peak system performance and 1000-fold increase in data volume; the next step is to enable MADmap to make effective use of the next generation of peta-scale HPC systems, with the particular scaling challenges they will present, in order to be able to analyze the next generation of CMB polarization experiments.

MADmap is a massively parallel implementation of a preconditioned gradient (PCG) solver for maximum likelihood CMB map-making under the assumption that the noise is Gaussian and piecewise stationary. Each datum is the sum of instrument noise and a sky signal (CMB+foregrounds)

$$d_t = n_t + s_t = n_t + P_{tp}\, s_p$$

where P_{tp} is the pointing matrix, giving the weight of each sky pixel p in sample t. Given the inverse time-time noise correlation matrix $N^{-1}_{tt'}$ the maximum-likelihood map m is then given by [29, 28]

$$m = \left(P^T\, N^{-1}\, P\right)^{-1} P^T\, N^{-1}\, d$$

Writing

$$\begin{aligned} M &= P^T\, N^{-1}\, P \\ b &= P^T\, N^{-1}\, d \end{aligned}$$

this can be cast in PCG form

$$Q^{-1}\, M\, m = Q^{-1}\, b$$

for some pre-conditioning matrix Q – typically chosen to be the trivially-invertible block-diagonal white noise approximation to M, with each 3×3 IQU-block constructed using just the diagonal of N^{-1}. Key to the MADmap implementation is to note that, by construction, N^{-1} is block band Toeplitz in the time domain (with each block corresponding to an interval over which the noise is stationary) and therefore diagonal in the Fourier domain. Multiplying a pixel-domain vector by M is therefore most efficiently performed by explicitly un-pointing the vector to the time-domain, Fourier transforming it, multiplying by the diagonal matrix, inverse Fourier transforming it, and re-pointing to the pixel-domain.

Until recently, the bottlenecks to scaling MADmap have been its IO requirements – specifically reading the time-ordered pointing and observation data. Like many massively parallel CMB codes, MADmap uses the M3 data abstraction layer [6, 17] to isolate the generic analysis algorithm from the particular details of a specific experiment's data format and distribution, since these are invariably unique to each experiment. Instead of individually recasting data to each application's preferred format and distribution, as used to be the norm, the M3 data abstraction layer provides a simple API for each CMB data type and uses an XML description of the data files included in a particular analysis (in a run configuration file, or RunConfig) to enable an application's generic data requests to be converted into the appropriate specific file operations. By virtue of its support of arbitrary data formats and distributions, M3 also allows application codes transparently to access compressed, multi-component or virtual data by uncompressing, combining or simulating the requested data on demand.

Historically, CMB experiments would reconstruct the pointing for every sample from every stream and save these on disk to be accessed by subsequent analyses. However this detector-specific full pointing came from first reconstructing the pointing of a single fiducial line-of-sight (such as the telescope boresight) from instruments like star trackers and gyroscopes that are sampled much less frequently than the detectors. Reducing the IO load of the pointing data has been achieved by using the Generalized Compressed Pointing (GCP) library which takes the sparse-sampled fiducial pointing and calculates the detector-specific full pointing over some specific interval only when the application code requests that data through the M3 interface. This compresses the pointing data volume by factors of (i) the number of detectors (10^3-10^4) and (ii) the ratio of the sampling rate of the

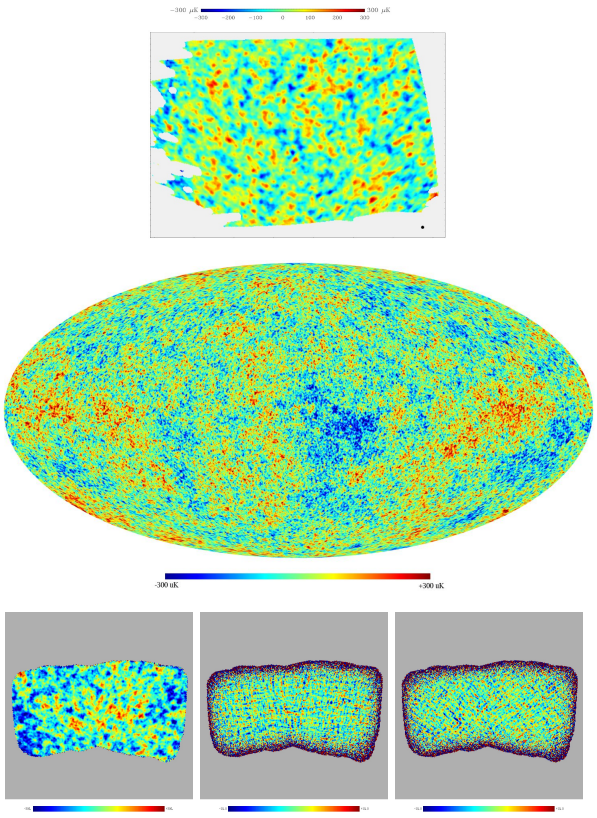

Figure 1: MADmap scaling to date: an O(10^8-sample, 10^5-pixel) BOOMERanG-98 temperature map calculated in 2000 on 128 Cray T3E processors, an O(10^{10}-sample,10^8-pixel) single-frequency simulated Planck temperature map calculated in 2005 on 6000 IBM SP3 processors, and an O(10^{11}-sample,10^5-pixel) simulated EBEX temperature and polarization map triplet calculated in 2008 on 15360 Cray XT4 cores.

detectors to the sampling rate of the pointing instruments (10^2-10^3), for an overall reduction of 5-7 orders of magnitude so that even the largest projected experiment only requires a few GB of pointing data.

Unlike the pointing data, real detector data do not have the kind of redundancy that allows for compression and reconstruction on the fly. However, by far the majority of the detector data used in CMB analyses is simulated – either for mission design studies, or as Monte Carlo realizations for power spectrum estimation. In these cases simulation is immediately followed by analysis, which has traditionally meant first performing the simulation and writing the detector data out to disk, and then reading those data back in for analysis. To reduce the IO load of such detector data the On-The-Fly Simulation (OTFS) library simulates an interval of a particular detector's data data only when requested by the analysis application, again through the M3 data abstraction interface. Since these data now need never touch disk, this approach can completely solve the simulated detector data IO problem for any application using the M3 interface, albeit at the cost of re-calculation for each re-analysis.

3.2 Communication Bottleneck

With the IO bottleneck solved, the next scaling issue to emerge has been in MADmap's communication requirements. Since the dominant computational cost scales with the number of time samples, load-balancing requires these time ordered data to be equally distributed over the processes. Given this data distribution, each process then has some of the information about some subset of the pixels (i.e. those pixels observed by its time-ordered data). At each PCG iteration, these pixel data must be reduced over all of the processes to generate the complete updated map.

The mapping between time-ordered data and sky pixels depends on the experiment's scanning strategy. Resolving CMB maps in the presence of non-white detector noise means that a fraction of the pixels in the map must be re-observed after some significant time has passed, and preferably in an orthogonal direction to the first scan. This technique is known as cross-linking, and it is precisely these re-observed pixels that end up being shared across the processes.

The required communication can most simply be performed with MPI's global reduction collective MPI_Allreduce, although often there is insufficient memory available to each process to store the entire map and the reduction must be buffered. Having chosen a supportable buffer size (n_b), a pipelined reduction on the entire map is performed with one call to MPI_Allreduce for each buffer. The buffer is filled with the value stored locally, or the identity of the reduction operator if no data is stored locally. The reduction is then performed, and each process copies the pertinent values out of the buffer into a local pixel vector before the operation is repeated.

We can utilize a simple network performance model to describe the complexity of this communication step. Assume that the time taken to send a message between two processes is given by $\alpha + n\beta$, where α is a term that accounts for the latency per message, β is the transfer time per byte (inverse of bandwidth), and n is the message size. This model does not account for network congestion, message sizes, or the network topology. However, prior work [13, 26, 21] suggests that this is a reasonable first-order approximation for comparing alternative algorithmic approaches.

Most current MPI implementations employ Rabenseifner's approach [25, 30] for MPI_Allreduce with large message sizes. This algorithm performs a reduce-scatter using the recursive-halving technique and then a gather operation using a binomial tree approach. For p processes, the complexity of this algorithm in terms of the α-β model is

$$2\log p\,\alpha + 2n\frac{p-1}{p}\beta$$

The bandwidth component of this algorithm is a notable reduction over an older binomial tree-based algorithm for Allreduce, which has a $n\log p\ \beta$ term. Our buffered reduction approach based on Rabenseifner's Allreduce algorithm would thus have a complexity of

$$2\frac{\mathcal{N}_p}{n_b}\log p\,\alpha + 2\mathcal{N}_p\frac{p-1}{p}\beta$$

Note that the volume of data exchanged in this approach may be much larger than necessary. If every process observes every pixel (i.e., we perform a reduction over a dense vector), then this approach would be optimal. On the other

hand if the number of pixels shared between processors is comparatively small (as is often the case), then much of the data communicated in this technique will be the identity operator for the reduction, which is wasteful. Although Rabenseifner's algorithm may not be used in all MPI implementations, any binomial tree-based reduction algorithm only changes our analysis by a potentially larger $n * \log p$ bandwidth component.

3.3 Collective Communication Optimization

The collective problem in MADmap can be generalized and formally stated as follows: each process i locally stores n_i key-value pairs, with $\Sigma_1^p n_i = n_{all}$. The keys are non-negative integers that lie in the range $[0, C]$. A global reduction operation with a binary associative operator is performed, but a process only needs to store the updated values of its local keys. Our objective is to minimize the total execution time for this collective operation, which is a sum of the local computation time and time spent in communication (which we will express in terms of the α-β model). Also, let o_{ij} denote the key counts shared by processes i and j, and $n_{lmax} = \max(n_i), 1 \leq i \leq p$.

This problem definition captures several common global communication patterns observed in parallel scientific computing. For instance, consider the Allreduce collective over a dense vector. In this case, $n_i = n_{lmax} = n_{all}/p = C$ for all $1 \leq i \leq p$ and $o_{ij} = C$ for all $1 \leq i, j \leq p$. Further, since keys are ordered in $[0, C]$ and sorted, they need not be explicitly represented. In this case, Rabenseifner's algorithm may be appropriate, as it achieves a good balance between the number of messages sent and the data volume exchanged over the network.

Nearest-neighbor communication would correspond to the case where o_{ij} is non-zero when $|i-j| = 1$, and 0 for all other $i \neq j$. Here, the optimal strategy would be to have neighbors exchanging the data they share and then performing reductions locally. The corresponding communication cost would be $2\alpha + 2n_{lmax}\beta$.

A third case that lies in between the dense vector reduction and nearest neighbor communication patterns is when each process shares data with a few other processes (say, more than one, but less than $\log p$). Hoefler et al. [14] study this problem and its variations, and design new, specialized collectives to address this problem. Hoefler and Träff [15] also make the case for better support of "sparse" communication patterns within MPI, where the sparsity refers to the number of communicating processes.

Clearly, the distribution of keys within $[0, C]$ for each process, the key overlap count between pairs of processes, the count of keys owned by each process, the values of C and n_{all}, all contribute to determining the optimal reduction strategy for a parallel platform.

To drive our design of a faster communication approach, we analyze the sparsity pattern exhibited by the sky pixel data partitioning . Figure 2 gives the count of sky pixels shared by one MPI process (process 0 in this case) with all other processes, for an experiment where $\mathcal{N}_p$ is 150 million. The data represents a distribution of pixels corresponding to a year of Planck simulated data using four 217 GHz channel detectors. Analysis shows that process 0 shares a significantly large fraction of its total number of local pixels (98.5%) with a few other processes (in this case, process IDs 1-20), and shares at least few pixels (say, < 1000) with al-

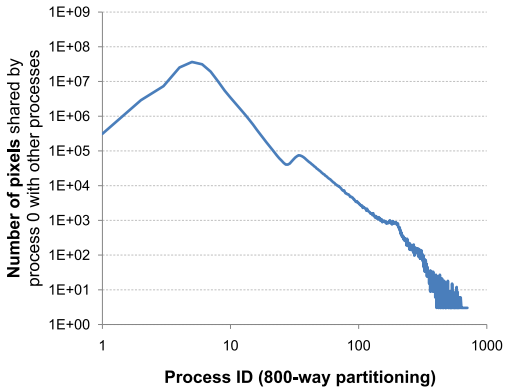

Figure 2: A depiction of the overlap pattern exhibited by a typical distributed sky-pixel data representation (Planck simulation data, $\mathcal{N}_p$ is 150 million).

most all other processes. Conversely, few pixels appear in the local partitions of almost all processes, and a majority of pixels appear in the local partitions of less than 20 processes. Additionally, the actual pixel integer identifiers that are local to each process are not contiguously ordered, i.e., process 0 contains roughly $\mathcal{N}_p/p$ pixels, but the pixel identifiers IDs are spread out in the entire range of $[0, \mathcal{N}_p]$. Restating these observations in terms of the collective problem definition, C would be equal to N_p, n_{all} is typically 2-6$\times$ the value of C, and the overlap counts and patterns are neither nearest-neighbor nor amenable to a sparse collective implementation. Furthermore, the values of C can be very large, and thus accumulation of all pixel identifiers and values on a single process may not possible.

Our new algorithm is primarily motivated by the potential data volume reduction that can be achieved by avoiding the buffered Allreduce approach. Note that n_{all} is not more than 6 times the value of N_p at process concurrencies up to 8000, and that the pixel counts per process are roughly the same. Furthermore, analysis of Figure 2 shows that Σo_{ij} is bounded by $2n_{lmax}$ for all the processes. Thus, if each process were to exchange only the data it shares with other processes and then perform the reduction locally, the bandwidth term in the α-β model would decrease from N_p to $2n_{lmax}$, which is almost a $\mathcal{O}(p)$ reduction.

This pairwise exchange scheme is implemented in a two-step approach. In the first "preprocessing" routine, each process determines the pixel identifiers (keys) that it shares with all other processes. Since the cumulative pixel overlap count per process is bounded, this information can be stored in memory for the duration of the simulation. Whenever a global reduction needs to be performed, each process only sends the values corresponding to the keys it shares with every other process. This is accomplished using a single call of the MPI_Alltoallv collective, which internally relies on pair-wise sends and receives to communicate the data.

Figure 3 presents the execution time of the various subroutines in the preprocessing step, on the Franklin system (described in Section 4). Our algorithm stores the local pixel identifiers in a sorted array on every process, and begins by determining n_{lmax}, which allows the estimate of n_{all} and the associated buffer sizes used for all the communication steps. Next, memory is allocate memory and initialize the data buffers to be exchanged (labeled as "Init buffer index"

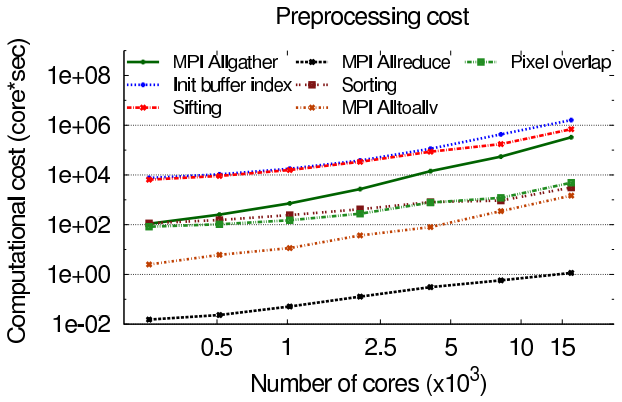

Figure 3: A breakdown of the local computation and communication components (on the Franklin system) in the preprocessing phase of the new collective algorithm.

in Figure 3). At this stage, we calculate all pairwise overlap counts and shared pixel locations. Exploiting symmetry, the pairwise count can be reduced from p^2 to $p(p-1)/2$. Instead of pairwise exchanges of data, the MPI_Allgather routine is utilized in a buffered fashion ("MPI Allgather" in Figure 3). This reduces the number of pairwise message exchanges, as MPI Allgather implementations typically utilize a tree-based algorithm. Each process inspects the pixel identifiers received from all other processes and notes the shared pixels. The local pixels are then stored in a Judy array data representation [3] to permit very fast membership queries. The step labeled "Sifting" in the figure captures the time taken for these lookups. Finally, the overlap pixels identifiers are stored compactly in a sorted array. The communication complexity of the preprocessing step is

$$\log p \frac{n_{lmax}}{n_b}\alpha + pn_{lmax}\frac{p-1}{p}\beta$$

in the α-β model (dominated by the MPI_Allgather step), and the local computation is $O(pn_{lmax})$ due to the initialization and sift operations. Figure 3 indicates that these three steps – Allgather, sift, and initialization – are indeed the steps that dominate the overall execution time. Future work will investigate additional performance optimizations such as simultaneous, asynchronous execution of the gather and sift steps using non-blocking collectives, and further optimization of the initialization routines.

Subsequent calls to a global reduction involve pairwise data exchanges utilizing the shared pixel information collected in the preprocessing routine. This step would be dominated by the cost of the Alltoallv routine: $p\alpha + kn_{lmax}\beta$, where k is a small number that bounds the overlap count in terms of n_{lmax}. The quadratic scaling of the number of messages exchanged may be a potential drawback of our scheme. All-to-all communication is a known bottleneck for MPI implementations and high concurrency runs [2]. However, this message count is inevitable if we rely on pairwise exchanges of data and try to minimize the total data volume. In future work, we will investigate approaches that would reduce the number of messages from p to a smaller, manageable value. For instance, the subset of pixels identifiers that appear on a majority of processes can be identified in the preprocessing step and reduced separately. This may necessitate an increase in the preprocessing step's local computation. Note that our approach does not directly utilize hardware-accelerated collective primitives for communication, but instead, optimizes code for compliant MPI functions. Our communication algorithm will automatically benefit from vendor-optimized MPI, which we expect to leverage hardware acceleration internally where available.

3.4 Threading Optimization

Our approach can also alleviate the communication cost by reducing the total number of MPI tasks p. To do so, we adopt a hybrid implementation that uses a single MPI task per socket or node, and as many OpenMP threads per task as there are cores socket/node. This hybrid approach required the implementation of new multithreaded routines for computationally demanding MADmap components, including FFTs (using the threaded ACML[1] or IMKL[18] libraries depending on their availability on a particular platform); the mapping of global to local pixel indices (using a threaded lookup with Judy arrays[3]); and the pointing matrix-vector multiplication, calculation and application of the pre-conditioner in the conjugate gradient calculation, calculation of the pixel overlap in the communication preprocessing step, and parts of the GCP and M3 libraries (all primary in their for-loops). OpenMP pragmas are used to parallelize the code, creating additional auxiliary data structures wherever required (such as thread-local copies for private buffers) in a space-efficient manner.

A few computational phases have not yet been threaded because of for-loops present in non-canonical form. These include inverse matrix multiplication, derivation of pixel data distribution, initializing data structures for convolution algorithm, calculation involved before writing output maps, and some parts of M3. Due to these unthreaded components, the cumulative computational time of threaded MADmap is currently 50% slower that the unthreaded case (i.e, with one MPI task per core). Implementing multithreaded versions of these remaining components as well as increasing the efficiency of threading in MADmap will be a key area of focus in our future work.

4. EXPERIMENTAL SETUP

To investigate the benefits of the new communication algorithm and hybrid programming in MADmap, we performed strong scaling experiments on a typical Planck-scale dataset on four different implementations of MADmap on five HPC systems, described in Section 4.2 — Franklin and Hopper at NERSC, Jaguar at Oak Ridge National Lab, and Pleiades-H and Pleiades-W at NASA Ames Research Center.

4.1 Data Analyses

MADmap is evaluated at eight different concurrencies ranging from 256 to 16384 cores, at powers of two for Franklin and Pleiades-H (with four cores per socket) and at the closest multiple of six to these for Hopper, Jaguar, and Pleiades-W (with six cores per socket). At each concurrency, we measured the overhead for communication, computation, and disk I/O incurred by MADmap when analyzing simulated data corresponding to a year of Planck observations of the entire sky from all 12 detectors at one of its key observing frequencies, so that ($\mathcal{N}_t \sim 7.5\times10^{10}$ and $\mathcal{N}_p \sim 1.5\times10^{8}$). All the runs were executed for 50 iterations of the PCG solver, a typical number for mapping polarized data, and to compare

Table 2: Highlights of CPU and node architectures for examined platforms, all evaluated processors are superscalar, out-of-order. MPI bandwidth is measured as the maximum MPI point-to-point bandwidth with message sizes from 4 Bytes to 256 MBytes. MPI latency is measured with 4-Byte messages.

Core Architecture	**AMD Budapest**	**AMD Istanbul**	**Intel Harpertown**	**Intel Westmere**	**AMD MagnyCours**
Clock (GHz)	2.30	2.6	3	2.93	2.1
DP Peak (GFlop/s)	9.20	10.4	12	11.72	8.4
Private L1 Data Cache	64 KB	64 KB	32 KB	32 KB	64 KB
Private L2 Data Cache	512 KB	512 KB	—	256 KB	512 KB
Socket/Node Architecture	**Opteron 1356 Budapest**	**Opteron 2435 Istanbul**	**Xeon E5472 Harpertown**	**Xeon X5670 Westmere**	**Opteron 6172 Magny-Cours**
Cores per Socket	4	6	4	6	12
Shared Cache per Socket	2 MB L3	6 MB L3	12 MB L2	12 MB L3	12 MB L3
Sockets per SMP	1	2	2	2	2
Node DP Peak (GFlop/s)	36.8	124.8	96	140.6	201.6
DRAM Pin Bandwidth (GB/s)	12.8	25.6	21.33	32.0	25.6
Node DP Flop:Byte Ratio	2.9	4.9	4.5	4.4	7.9
System Architecture	**Cray XT4 Franklin**	**Cray XT5 Jaguar**	**Intel Cluster Pleiades-H**	**Intel Cluster Pleiades-W**	**Cray XE6 Hopper[1]**
Interconnect	Seastar2 3D Torus	Seastar2+ 3D Torus	Infiniband DDR	Infiniband QDR	Gemini 3D Torus
Total Nodes	9,660	18,688	5,888	2,048	6,392
Peak Bandwidth (GB/s per direction)	3.8	4.8	2.5	5	4.8
Measured MPI point-point bandwidth (GB/s)	1.65	1.6	1.66	3.13	5.95
Measured MPI latency (μs)	8.15	8.26	1.32	1.83	1.4
Compiler vendor, version	PGI 10.1	PGI 10.3	Intel 11.1	Intel 11.1	PGI 10.9
MPI Vendor, MPT version	Cray 4.0.3	Cray 4.0	SGI 1.25	SGI 1.25	Cray 5.1.2

performance of four different MADmap versions.

Unthreaded Allreduce: The existing (original) version of MADmap that uses MPI_Allreduce for communication. This application is not threaded and runs with one MPI process per core.

Unthreaded Allgather: This version of MADmap replaces MPI_Allreduce with the new collective communication algorithm discussed above, but without threading (one MPI task per core). Since the cost of our new algorithm is dominated by the MPI_Allgather in the preprocessing step, we refer to this as Allgather-based approach. Evaluating this version at various concurrencies addresses the question of whether the new algorithm by itself is sufficient to eliminate the high concurrency communication bottleneck.

Threaded Allreduce: In this version, existing implementation of MADmap is threaded using OpenMP pragmas. Evaluating this version tells us whether reducing the number of communicating processes with threading alone can provide a solution to the communication bottleneck at high concurrencies.

Threaded Allgather: This version includes both the optimized communication algorithm as well as OpenMP threading, and quantifies the impact of including both the optimization strategies.

Note that both Threaded Allreduce and Threaded Allgather run with one MPI task per socket, combined with as many OpenMP threads as there are cores in that socket.

4.2 Architectural Platforms

To evaluate MADmap in a range of HPC environments we conduct our experiments on five large-scale HPC platforms, the Cray XT4, XT5 and XE6 and two generations of Intel/Infiniband clusters. The Cray XT is designed with tightly integrated node and interconnect fabric, opting for a custom network ASIC and messaging protocol coupled with a commodity AMD processor. In contrast, the Intel/IB cluster is assembled from off-the-shelf high-performance networking components and Intel server processors.These represent common design trade-offs in the high performance computing arena. Table 2 shows architectural highlights of the examined platforms.

Franklin: Cray XT4: Franklin, a 9,660 node Cray XT4 supercomputer, is located at Lawrence Berkeley National Laboratory (LBNL). Each XT4 node contains a quad-core 2.3 GHz AMD Opteron processor, which is tightly integrated to the XT4 interconnect via a Cray SeaStar2 ASIC through a HyperTransport (HT) 2 interface capable of 6.4 Gbyte/s. All the SeaStar routing chips are interconnected in a 3D torus topology with each link is capable of 7.6 Gbyte/s peak bidirectional bandwidth, where each node has a direct link to its six nearest neighbors. Typical MPI latencies will range from 4.5 - 8.5 μs, depending on the size of the system and

[1]**To reviewers and PC members:** the Hopper system is still under evaluation, thus the presented Hopper data is currently confidential. Hopper will be accepted in time for the final paper version, allowing public distribution of its performance data.

the job placement. The Opteron Budapest processor is a superscalar out-of-order core that may complete both a single instruction-multiple data (SIMD) floating-point add and a SIMD floating-point multiply per cycle, the peak double-precision floating-point performance (assuming balance between adds and multiplies) is 36.8 GFlop/s. Each core has both a private 64 Kbyte L1 data cache and a 512 Kbyte L2 victim cache. The four cores on a socket share a 2 Mbyte L3 cache. Unlike Intel's older Xeon, the Opteron integrates the memory controllers on chip and provides an inter-socket network (via HT) to provide cache coherency as well as direct access to remote memory. This machine uses DDR2-800 DIMMs providing a DRAM pin bandwidth of 12.80 Gbyte/s per socket.

Jaguar: Cray XT5: The 18,688 node Jaguar XT5 platform is currently the number two system on the TOP500 [31]. This successor to the XT4 line contains nodes with two hexa-core 2.6 GHz AMD Opteron processors, and a Cray SeaStar2+ 3D torus interconnect capable of 9.6 Gbyte/s. Typical MPI latencies will range from 4.5 - 8.5 μs, depending on the size of the system and the job placement. The next-generation Opteron Istanbul processor has a larger 6MB semi-exclusive L3 cache and a peak theoretical node rate of 124.8 double-precision GFlop/s. To mitigate snoop effects and maximize the effective memory bandwidth, Istanbul uses 1MB of each 6MB cache for HT assist (a snoop filter). The snoop filter enables higher bandwidth on large multi-socket SMPs.

Pleiades-H: Intel Harpertown Cluster: The 9,216 node Pleiades cluster, located at NASA Ames Research Center consists of three Xeon-based clusters, two of which are examined in this study. The Pleiades-H cluster consists of 5,888 dual-socket nodes utilizing 3.0 GHz quad-core Intel Harpertown processors, connected via a DDR IB network in partial 11D hypercube. Providing an interesting comparison to the Opterons, the Xeon E5472 (Harpertown) uses a modern superscalar out-of-order core architecture coupled with an older frontside bus (FSB) architecture in which two multichip modules (MCM) are connected with an external memory controller hub (MCH) via two frontside buses.Unfortunately the limited FSB bandwidth (10.66 Gbyte/s) bottlenecks the substantial DRAM read bandwidth of 21.33 Gbyte/s (subsequently released Nehalem processors have abandoned the front-side bus in favor of on-chip memory controllers). Each core runs at 3 GHz, has a private 32 KB L1 data cache, and, like the Opteron, may complete one SIMD floating-point add and one SIMD floating-point multiply per cycle. Unlike the Opteron, the two cores on a chip share a 4 Mbyte L2 and may only communicate with the other two cores of this nominal quad-core MCM via the shared frontside bus.

Pleiades-W: Intel Westmere Cluster: Our study also examines performance on the Pleiades-W cluster that consists of 2,048 dual-socket nodes containing 2.93 GHz octal-core Intel Westmere processors connected via a QDR IB network. This recently released design the latest enhancement to the Intel "Core" architecture, and represents a dramatic departure from Intel's previous multiprocessor designs. It abandons the front-side bus (FSB) in favor of on-chip memory controllers. The resultant QuickPath Interconnect (QPI) inter-chip network is similar to AMD's HyperTransport (HT), and it provides access to remote memory controllers and I/O devices, while also maintaining cache coherency. Each core has a private 256 KB L1 and a 1 MB L2 cache, and each socket instantiates a shared 12 MB L3 cache. Additionally, each socket integrates three DDR3 memory controllers providing up to 32 GB/s of DRAM bandwidth to each socket.

Hopper: Cray XE6: The 6392 node Hopper cluster debuted this year as the fifth fastest system on Top 500 list. Each node contains two 12-core 2.1 GHz AMD 'Magny-Cours' processors and a Cray Gemini interconnect with an effective bandwidth of 168 GB/s. Each processor has 2 dies with 6 cores on each die. Each die has 2 memory channels and is a NUMA node. There are 4 HyperTransport 3 (HT3) links per processor providing a peak bandwidth of 25.6 GB/s per processor. Each core has private L1 and L2 caches with 64KB and 512KB respectively and a L3 cache of 12 MB shared between the two dies in a processor. The processors are interconnected in a 3D torus topology with each node providing an aggregate bandwidth of 168 GB/s.

5. RESULTS AND DISCUSSION

For each code configuration and concurrency, we time the calculation, communication and IO components, across all evaluated HPC systems — reporting our results in total core-seconds; a constant value therefore represents perfect scaling for this problem. Figure 4 shows the performance of Unthreaded Allreduce, Unthreaded Allgather, Threaded Allreduce and Threaded Allgather on Franklin, Jaguar, and Pleiades-H and -W[2]. Each column compares the performance of the various implementations on a single system and each row compares the performance of a particular implementation across the systems. The threaded implementations are run with one process per socket and as many threads as there are cores per socket; in addition, for all of the dual-socketed systems use the Threaded Allgather approach alone via a single process per node with as many threads as cores on the node.

5.1 Comparison Across Implementations

Results show that on all systems, the communication bottleneck emerges in the Unthreaded Allreduce, with communication dominating above a few thousand cores. The Unthreaded Allgather approach increases the concurrency at which communication become a bottleneck. But once it does, its cost rapidly comes to exceed that of the Unthreaded Allreduce. In both of the threaded cases, the communication cost is reduced, marginally for the Allreduce and significantly for the Allgather. Both of these observations are consistent with our complexity analyses, in which the Allreduce latency depends on $\log p$, and the Allgather latency on $p \log p$ and the transfer time on p. At a concurrency of 16K processors, the improvement in communication overhead is maximum on Jaguar with a speedup of 14.2× using threaded Allgather. On Franklin, Hopper, Pleiades-W, and Pleiades-H the communication speedups are 1.95×, 2.35×, 3.44× and 4.65× respectively. At the same time, the calculation cost increases due to the incomplete threading — as expected from Amdahl's Law — and the IO cost decreases, since much of the data ingest is implemented by process 0 performing a serial read and broadcasting, now to a smaller number of processes.

[2]To date we have been unsuccessful in completing 16K-way unthreaded allgather run on Pleiades-H, and are working with Nasa Ames staff to resolve this in time for publication

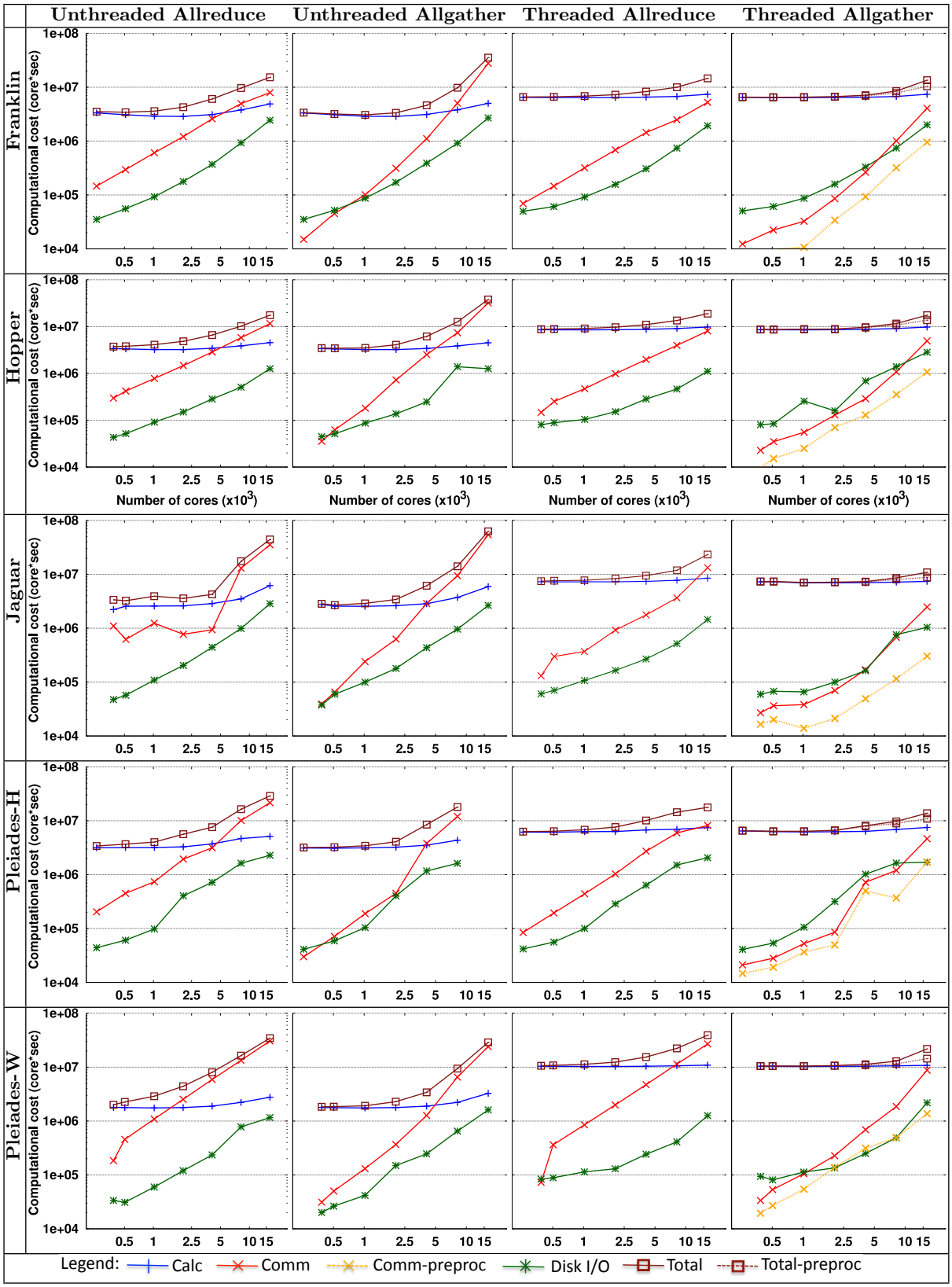

Figure 4: MADmap scaling runs on different platforms. Threaded Allgather and Threaded Allreduce execute with one process per socket with four or six threads. The dotted lines in Threaded Allgather represent the total and communication time without the preprocessing overhead.

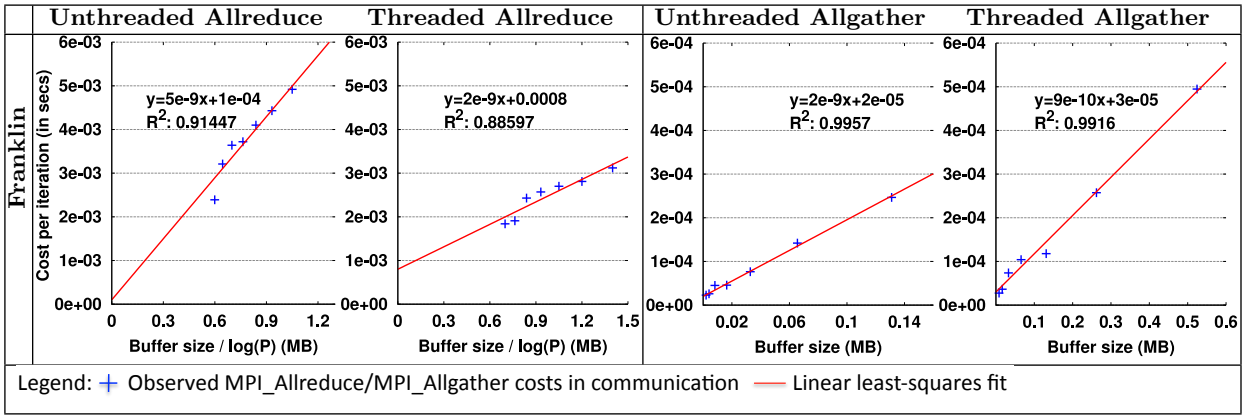

Figure 5: A least-squares fit of the observed Franklin communication costs (scatter data) for MPI_Allgather and MPI_Allreduce, to the linear α-β model expressions. The R^2 value indicates the deviation of the observed data from the fit, with a value close to 1 indicating a good fit.

To check whether our theoretical expressions for the communication costs are indicative of observed execution times in practice, we perform a least-squares fit of the observed data to linear model. Figure 5 depicts the MPI_Allreduce and MPI_Allgather execution times per iteration and the calculated trend line. For MPI_Allgather, we fit the data to the complexity cost of a ring-based (not tree-based) algorithm, since the MPI implementations employ a ring-based algorithm for the buffer sizes in our experiments. Observe from Figure 5 that the linear α-β model is a reasonable fit for MPI_Allreduce, and a very good fit for MPI_Allgather. The slope of the line corresponds to β in our model, which is the inverse of the sustained per-node network bandwidth for the collectives. Examining these values shows that the sustained bandwidth per process for the threaded case is greater by a factor of two for both MPI_Allreduce and MPI_Allgather. The measured point-to-point bandwidth and the small message latency (see Table 2) are upper and lower bounds respectively for the empirically-determined α and β values, thereby providing validation of our performance model. The impact of latency term is negligible in the MPI_Allreduce cost, due to the large message sizes and the tree-based algorithm employed. To summarize, we observe a sustained bandwidth of 0.86 GB/s for threaded Allreduce, 0.25 GB/s for unthreaded Allreduce, 1.15 GB/s for threaded Allgather and 0.58 GB/s for unthreaded Allgather. The communication costs on the remaining systems can be similarly analyzed, to estimate the potential for improvement in performance of the MPI collectives, and will be the focus of future investigations.

For the four evaluated systems with dual-socket nodes, threading per node performance can be compared against threading per socket. Since the former halves the number of MPI tasks, one might expect node-level threading to result in a similar reduction in the communication cost. However, at higher concurrencies, the communication cost actually increases on Jaguar and Pleiades-H, and stays roughly constant on Pleiades-W. This is likely because a single process per node is insufficient to saturate the available network injection bandwidth on Jaguar and Pleiades-H. Results also show the calculation cost roughly doubling due to the partial threading of the application and NUMA effects of a single process running on two sockets. Therefore, in our current implementation, threading beyond the socket level does not gain any additional benefit.

5.2 Comparison Across Systems

The communication cost for Unthreaded Allreduce at 16K processors on Jaguar, Pleiades-W, Pleiades-H and Hopper are 4.45, 3.85, 2.71 and 1.5 times the cost on Franklin respectively. The reason for this slowdown is due to the increased number of communicating processes per node on these systems: 4, 12, 12, 8 and 12 for Franklin, Jaguar, Pleiades-W, Pleiades-H, and Hopper, respectively. As a result, the per-node bandwidth is shared by all the processes per node on these systems, although Hopper has roughly three time the bandwidth per node compared to the other systems and about 1.5 times the bandwidth per node compared to Pleiades-W. Also, since the messages from multiple processes is serialized at the network interface of the node, increasing the number of MPI tasks per node limits scalability. In addition, variability due to other factors such as the type of interconnect used (Seastar2 or Infiniband), its configuration (3D Torus or Hypercube), and MPI implementation details also influence the communication costs.

At a concurrency of 16K in threaded MADmap, Franklin and Pleiades-H use 4096 MPI processes whereas Jaguar, Pleiades-W and Hopper use only 2730 processes. Threaded Allreduce reduces the communication overhead by 1.51, 2.66, 2.64, 1.12 and 1.44 with respect to their unthreaded counterparts on Franklin, Jaguar, Pleiades-H, Pleiades-W, Hopper respectively at a concurrency of 16K processors. As seen in the unthreaded case, the cost of Threaded Allreduce on Jaguar, Pleiades-H, Pleiades-W and Hopper is 2.54, 1.55, 5.15 and 1.54 times slower than Franklin. Although the total number of processes compared to Franklin is lower in Jaguar, each node has two MPI processes that share the inter-node network bandwidth; whereas Franklin has just one process per node. Compared to Franklin, the data per process on Jaguar is more, which in turn increases the number of communicating MPI tasks (since the message size is maintained constant). Unlike Jaguar and Pleiades-W that have two MPI processes sharing the node bandwidth, Hopper has four MPI process per node. Thus with even with more processes per node, the communication cost on Hopper is lower than all the other two sockets per node sys-

tems due to the availability of high bandwidth per node. Threaded Allgather further reduces the communication as the expensive preprocessing cost is incurred only once. At a concurrency of 16K processors, the communication cost for Threaded Allgather on Jaguar is 3.44 times faster than Franklin. The main reasons for this improvement is that the overhead of per-iteration communication is relatively inexpensive. The communication cost on Pleiades-H is comparable to the cost on Franklin as they use same number of total MPI processes. The marginal increase in cost in Pleiades-H can be attributed to the overhead due to serialization of messages at the network interface from two processes in a single node. On Pleiades-W, due the undetermined variability in the MPI implementation, the MPI_Allgather cost in preprocessing increases steeply at higher concurrency, thereby increasing the overall cost of communication. Additionally, Figure 4 shows that the communication cost on Hopper is comparable to Franklin and Pleiades-H.

The calculation cost with 16K processors for unthreaded MADmap are comparable on Franklin, Jaguar, Hopper and Pleiades-H. On Pleiades-W the application runs almost twice as fast compared to Franklin, Jaguar, Hopper and Pleiades-H respectively, as seen in Figure 4. Although Pleiades-H has a faster processor clock frequency compared to Pleiades-W, the bus to core ratio in Pleiades-W in 22 compared to 7.5 in Pleiades-H — account for its faster calculation cost, as higher ratio indicates greater bandwidth to transfer data. In addition the Pleiades-W processors support a maximum turbo frequency of 3.3GHz along with a Quick Path Interconnect (QPI) rate of 6.4GT/s compared to a front side bus frequency of 1600MHz in Pleiades-H.

However, the calculation cost for threaded MADmap on Pleiades-W is highest relative to the other platforms, including Pleiades-H The reason for this slowdown is likely due to different instruction set used in the compilation of application binary. Threaded MADmap was compiled on Pleiades-H with SSE4.1 instruction set whereas unthreaded MADmap uses the application binary compiled on Pleiades-W with the SSE4.2 instruction set. We could not successfully execute the threaded binary compiled with SSE4.2 on Pleiades-W and are investigating the problem. Finally, note that in all cases, the calculation cost remains roughly constant across all concurrencies, indicating a near-linear speedup.

5.3 Monte Carlo Analyses

While these results focus on a single analysis, actual CMB data computations are dominated by generating (simulating and mapping) sets of hundreds to tens of thousands of Monte Carlo data realizations. For these large-scale experiments, our methodology allows a single, off-line preprocessing step once — since the communication pattern will be common to all the realizations. Figure 6 shows the speedup achieved by the Threaded Allgather on all the platforms when the preprocessing is excluded, showing the overall speedup (written for each platform) as well as highlighting the dramatic reduction of communication overhead. Although the calculation costs have increased in the threaded code, this is more than offset by the communication improvements, and will be further optimized in subsequent studies. The overall speedups attained are 4.14×, 2.88×, 2.41×, 1.48×, 1.27× for Jaguar, Pleiades-H, Pleiades-W, Franklin and Hopper (respectively). The (preprocessing amortized) communication speedups now account for only 2% — 4% of the overall

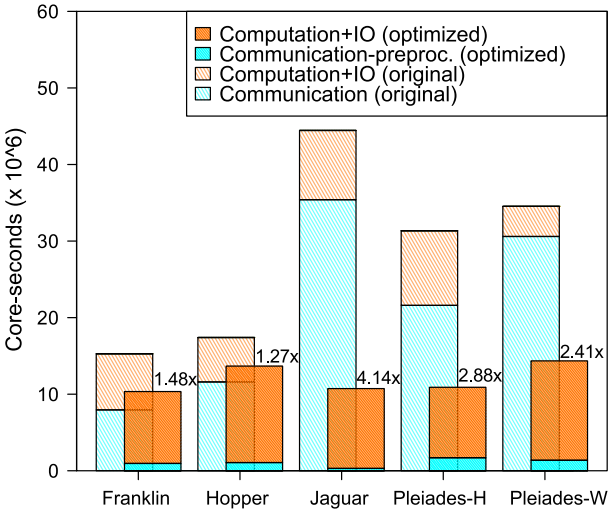

Figure 6: The speedup achieved using our new communication optimizations and inter-socket threading on the five different parallel platforms and 16K core concurrency. 'Original' and 'optimized' correspond to unthreaded Allreduce and threaded Allgather respectively. The overall speedup value is indicated above the optimized implementation bar. The new communication algorithm preprocessing time is not included in the cost of the optimized implementation, as it is amortized over 50 iterations.

runtime, showing impressive speedups compared to the original version of up to 116× on Jaguar, with improvements of 8.3×, 22.3×, 12.7×, and 10.8× on Franklin, Pleiades-W, Pleiades-H and Hopper (respectively).

6. CONCLUSIONS

Observations of the CMB have the potential profoundly effect on our understanding of the Universe and hence of fundamental physics at the highest energies. However, the exponential growth in the size of CMB datasets over the next 15 years means that our analyses have to stay on the bleeding edge of high performance computing for the next 10 epochs of Moore's Law. At present, the state-of-the-art in simulating and mapping CMB data sets is the MADmap code, but it suffers from a serious communication bottleneck when using more than a few thousand MPI tasks.

In this work we presented a two step approach to alleviate the communication bottleneck. The first step involves using a new algorithm that minimizes the communication data volume by replacing global reductions that include transfers of large amounts of redundant data with pairwise point-to-point communication, which only includes data that must be communicated. The second step leverages the OpenMP/MPI hybrid programming model, reducing the number of MPI tasks to only one per socket. Our work presents an extensive performance evaluation of these steps (both individually and in concert) across a wide range of large-scale HPC platforms and shows that at high concurrencies, our combined methodologies result in significant improvement of communication overhead — with reductions of one to two orders of magnitude compared with the original approach. As a result of these optimizations, the communication cost is no longer the bottleneck, thus causing

the calculation phase to emerge as the dominant overhead component. This bottleneck shift is well suited for next generation supercomputers whose computational throughput is expected to grow faster than interconnect messaging speeds. Future work will focus on effectively threading the remaining unthreaded computations, as well as exploring the potential of using accelerator-based systems such as GPUs to reduce the computation time.

Overall, this work enables CMB data analysts to take advantage of the largest peta-scale HPC systems, which will be essential to achieve the full scientific potential of the coming generation of B-mode CMB experiments.

7. ACKNOWLEDGEMENTS

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and the NASA Advanced Supercomputing facility at the Ames Research Center. This work was supported by the NSF PetaApps program under grant AST-0905099.

8. REFERENCES

[1] AMD Core Math Library, 2010. `http://www.amd.com/acml`.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J.L. Träff. Mpi on a million processors. In *Proc. 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30. Springer-Verlag, 2009.

[3] D. Baskins. Judy arrays web page, 2004. `http://judy.sourceforge.net/`.

[4] J. Bock et al. The experimental probe of inflationary cosmology (EPIC): A mission concept study for NASA's Einstein inflation probe. `http://arxiv.org/abs/0805.4207v1`, 2008.

[5] J.R. Bond, A.H. Jaffe, and L. Knox. Estimating the power spectrum of the cosmic microwave background. *Phys. Rev. D*, 57(4):2117–2137, 1998.

[6] C.M. Cantalupo, J.D. Borrill, A.H. Jaffe, T.S. Kisner, and R. Stompor. MADmap: A Massively Parallel Maximum Likelihood Cosmic Microwave Background Map-maker. *The Astrophysical Journal Supplement Series*, 187:212, 2010.

[7] CMBPol mission concept study, 2010. `http://cmbpol.uchicago.edu/`.

[8] COBE, 2010. `http://lambda.gsfc.nasa.gov/product/cobe`.

[9] S. Dodelson. *Modern Cosmology*. Academic Press, 2003.

[10] J. Dunkley et al. Five-year wilkinson microwave anisotropy probe observations: Likelihoods and parameters from the wmap data. *The Astrophysical Journal Supplement Series*, 180(2):306, 2009.

[11] G.H. Golub and C.F. Van Loan. *Matrix computations*. Johns Hopkins University Press, 3rd edition, 1996.

[12] E. Hivon et al. MASTER of the cosmic microwave background anisotropy power spectrum: A fast method for statistical analysis of large and complex cosmic microwave background data sets. *The Astrophysical Journal*, 567(1):2, 2002.

[13] R.W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[14] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proc. 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)*, pages 159–168. ACM, 2010.

[15] T. Hoefler and J.L. Träff. Sparse collective operations for mpi. In *Proc. 14th Int'l. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS09)*, 2009.

[16] E. Komatsu et al. Five-year wilkinson microwave anisotropy probe observations: Cosmological interpretation. *The Astrophysical Journal Supplement Series*, 180(2):330, 2009.

[17] M3 data abstraction library, 2010. `http://crd.lbl.gov/~cmc/M3/`.

[18] Intel Math Kernel Library, 2010. `http://software.intel.com/en-us/intel-mkl`.

[19] The OpenMP API specifications, 2010. `http://openmp.org/wp`.

[20] P. Oxley et al. The EBEX experiment. *Proc. SPIE*, 5543(1):320–331, 2004.

[21] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[22] Planck science team home, 2010. `http://www.rssd.esa.int/index.php?project=PLANCK`.

[23] PolarBeaR, 2010. `http://bolo.berkeley.edu/polarbear`.

[24] QUIET, 2010. `http://quiet.uchicago.edu`.

[25] R. Rabenseifner. New optimized MPI reduce algorithm, 2010. `https://fs.hlrs.de/projects/par/mpi/myreduce.html`.

[26] R. Rabenseifner and J.L. Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Proc. Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 309–335, 2004.

[27] G.F. Smoot et al. Preliminary results from the COBE differential microwave radiometers - large angular scale isotropy of the cosmic microwave background. *Astrophysics Journal*, 371:L1–L5, 1991.

[28] R. Stompor et al. Making maps of the cosmic microwave background: The MAXIMA example. *Phys. Rev. D*, 65(2):022003, 2001.

[29] M. Tegmark. CMB mapping experiments: A designer's guide. *Phys. Rev. D*, 56(8):4514–4529, 1997.

[30] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *Int'l. Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[31] Top500 supercomputer sites, 2010. `http://top500.org`.

[32] WMAP, 2010. `http://map.gsfc.nasa.gov`.