Parallelization of GSL on Clusters of Symmetric Multiprocessors

J. Aliaga, J.M. Badía, S. Barrachina, M. Castillo, R. Mayo, E.S. Quintana & G. Quintana

Univ. Jaime I, Castellón

F. Almeida, V. Blanco, C. Rodríguez, F. de Sande & A. Santos

Univ. de La Laguna

Notes

In this talk I will describe two parallel algorithms implemented using n hybrid MPI plus OpenMP programming model.

I will compare these algorithms with their versions implemented using a pure message-passing model.

1 Hybrid Architectures

Definition

Parallel architecture that combines distributed memory between different nodes with shared memory within each node.

- Mail Goal

  Searching scalability with the available technology and programming models

- Examples
  - Clusters of Symmetric Multiprocessors (SMP)
  - Large supercomputers: Earth Simulator, Mare Nostrum, etc.

Notes

I will use the following simple definition of hybrid parallel architecture. "A hybrid parallel architecture combines distributed memory between different nodes with shared memory within each node."

The main goal of using this kind of architectures is try to obtain the maximum scalability by using the available technology and programming models.

The number of hybrid architectures is clearly increasing, and they include an important part of the current clusters of personal computers in which each node contains more than one processor.

Hybrid architectures also include many of the larger supercomputers, like the Earth simulator or the Mare Nostrum supercomputers.

1 Hybrid Architectures (II)

Programming Models

- Data Parallelism: HPF
- Message Passing Model: Pure MPI
- Hybrid Model: Message Passing + Shared Memory
  - Message passing **between** nodes: MPI
  - Shared memory **within** node: OpenMP

Notes

We can use different programming models to program hybrid architectures:

For example, we can use some compiler based on a data parallelism model, like for example High Performance Fortran.

We can program hybrid architectures using a pure message-passing model and considering all the processors at the same level.

However, the most natural way to program a hybrid architecture seems to use a hybrid programming model. In this case a message-passing model is applied between the different nodes, and a shared memory model is applied within each node.

Trying to exploit the current standards, we have used MPI to apply the message-passing model and OpenMP to apply the shared memory model.

1 Hybrid Architectures (III)

First Approach

- Parallelization of two simple algorithms:
  - Sparse matrix-vector product
  - Dynamic programming problem
- Two schemes of parallelism
  - Embarrasingly parallel computation: Single computation step + single communication step
  - Synchronous computation: Multiple computation + communication/synchronisation step

Notes

In our first approach to apply the hybrid programming model we have solved two simple problems: the sparse matrix-vector product and the dynamic programming problem.

Each of these problems uses a different scheme of parallelism:

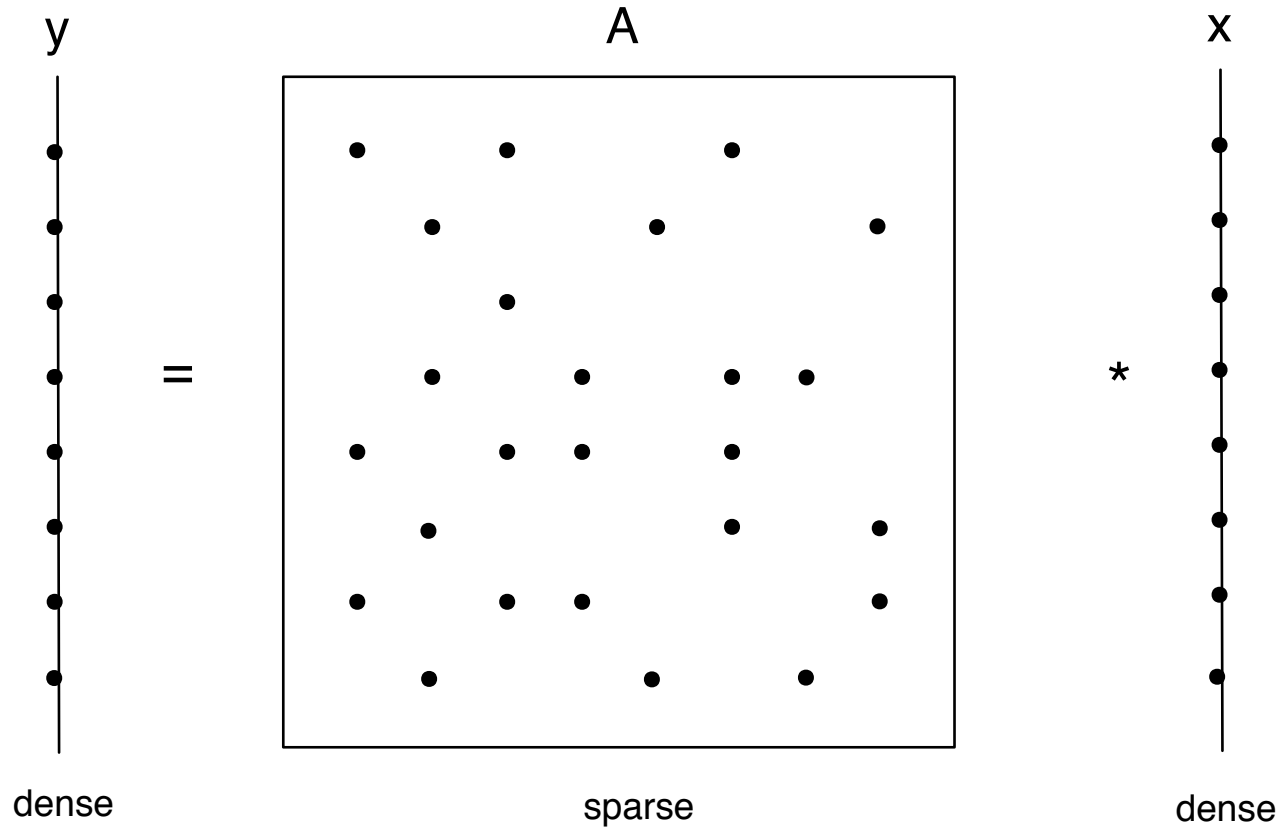
The sparse matrix-vector product uses what we can call embarrasingly parallel computation. In this case the parallel solution includes a first step of fully parallel computation without communications and a second step of communications.

On the other hand, the solution of the dynamic programming problem uses what is called a synchronous computation. In this case the parallel algorithm performs several iterations. Each iteration includes a computation step and a synchronization step based on collective communications.

Next I will describe now the parallelization of these two problems.

2 Sparse matrix-vector product

Basic sparse linear algebra operation arising in multiple areas: structural analysis, pattern matching, control of processes, tomography, etc.



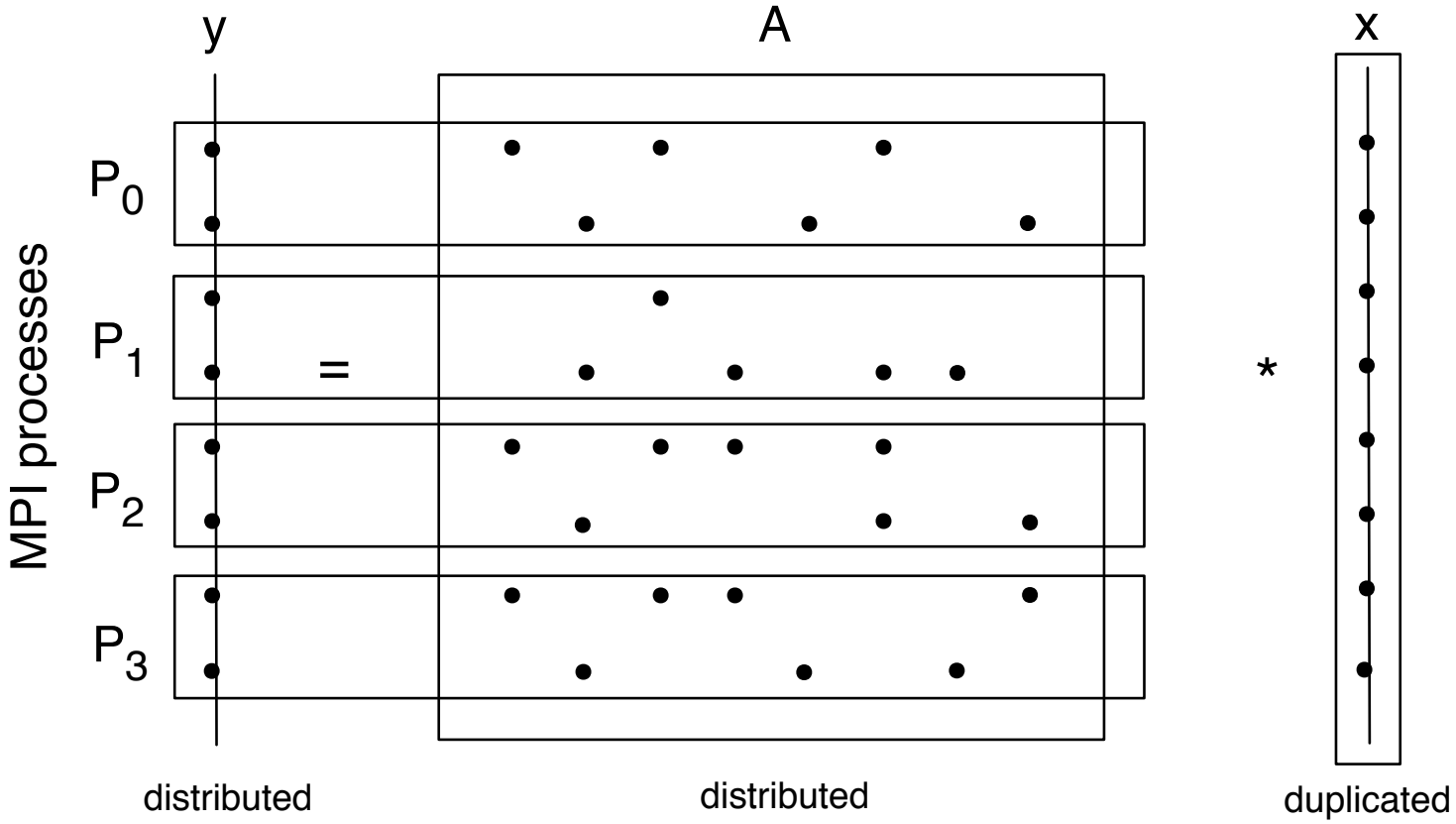
Notes

The sparse matrix-vector product is a well know operation that arises in multiple areas, including for example structural analysis, pattern matching, control of processes and many others.

In this problem, a sparse matrix A is multiplied by a dense vector x, and the solution is stored in another dense vector y.

2 Sparse matrix-vector product (II)

Rowwise parallelization. Pure MPI version



Notes

There are different parallel implementations of the sparse matrix-vector product. We have implemented a rowwise parallelization. In this algorithm we exploit the fact that different elements of the solution vector can be computed fully in parallel as dot products of different rows of the matrix and the vector x.

In a Pure MPI version of the product we can distribute different blocks of rows of the matrix to different MPI processes and duplicate the vector x on each process.

Each process computes fully in parallel a block of the solution vector. If we want to duplicate all the solution in all the processes, once the computations are finished we have to perform an allgather operation.

This version of the product can be executed on a distributed memory architecture with one processor per node or in a hybrid architecture with several processors per node. In the hybrid case, we can exploit the hybrid architecture by using an specific version of the MPI library that performs the communication among the processes using the communication network between the nodes or the shared memory within each node.

2 Sparse matrix-vector product (III)

Pure MPI code

```
for (i = 0; i < local_size1; i++) { // local dot products
   pos = rowptr[i];  // first element in row i
   k = row_index(pos);
   temp = 0.0;
   for (j =  0; j < nz_in_row(i); j++) {
     temp += value_in(pos) * x[k];
     inc_coordinate(pos);
   }
   y[k] += alpha * temp;
 }
 MPI_Allgather(...); // Result gathering
```

Notes

In this slide you can see the code of the Pure MPI version of the algorithm executed on each process.

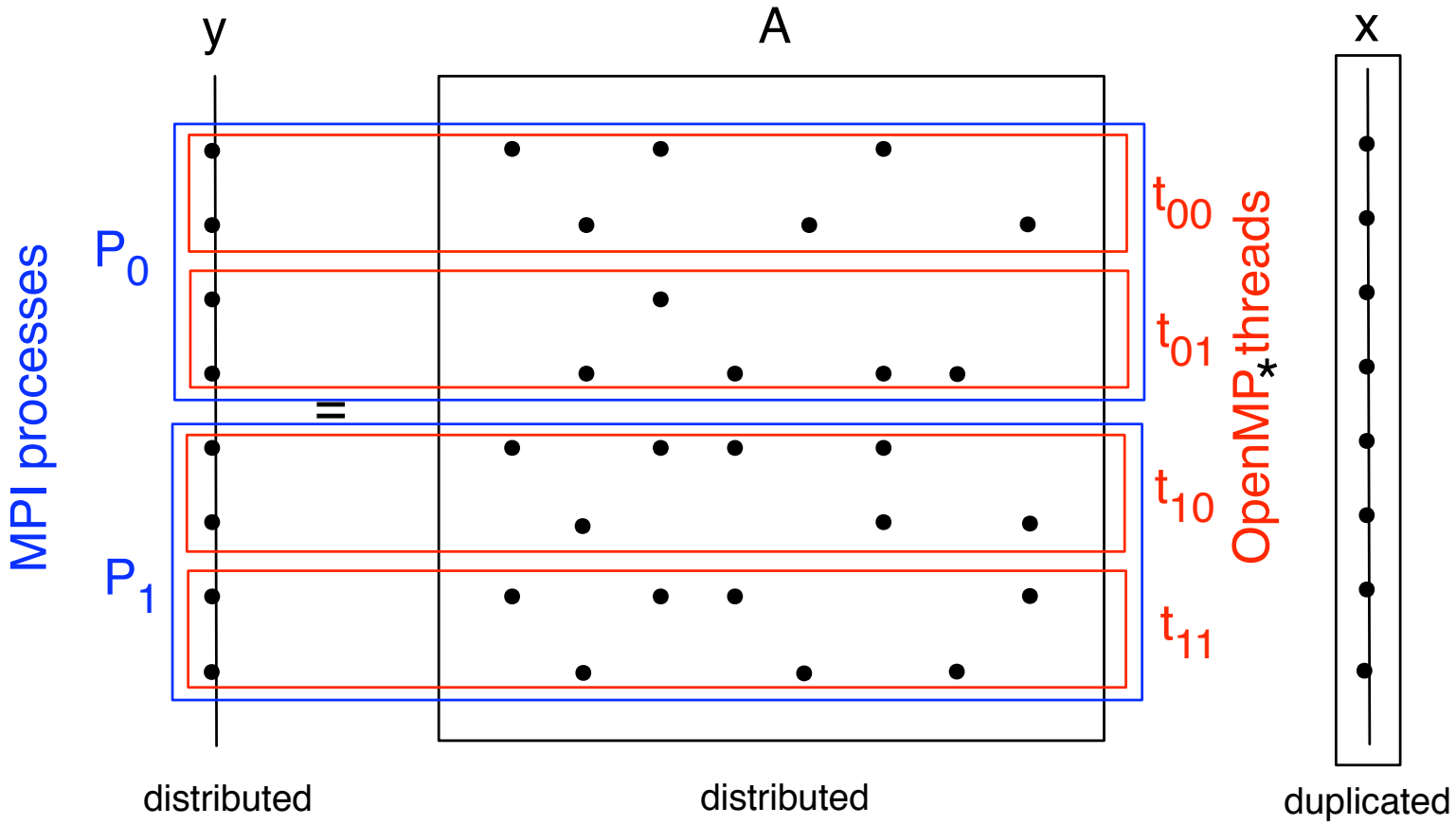
We can see the two basic steps of the algorithm.

During the first step, in the for loop, the different MPI processes perform fully in parallel their local dot products and compute a block of the solution vector.

The second step includes the communication operation that gathers the solution vector in all the processes.

2 Sparse matrix-vector product (IV)

Hybrid MPI + OpenMP version



Notes

In order to implement the hybrid version of the algorithm we exploit the same kind of parallelism than before at two different levels.

In the first level we distribute different blocks of rows of the matrix to different MPI processes. For example two processes like in the slide. Each process is executed in a different node.

In the second level, each MPI process distributes the computation of its local dot products to different OpenMP threads. In the example, each MPI process executes 2 OpenMP threads. Each thread is executed in a processor of the node exploiting its shared memory architecture.

2 Sparse matrix-vector product (V)

Hybrid MPI + OpenMP code

```
#pragma omp parallel for private (pos, k, temp, j) schedule(...)
  for (i = 0; i < local_size1; i++) { // local dot products
    pos = rowptr[i];  // first element in row i
    k = row_index(pos);
    temp = 0.0;
    for (j =  0; j < nz_in_row(i); j++) {
      temp += value_in(pos) * x[k];
      inc_coordinate(pos);
    }
    y[k] += alpha * temp;
  } // end of #pragma
  MPI_Allgather(...); // Result gathering
```

Notes

The hybrid code of the sparse matrix-vector product is basically the same that the code of the Pure MPI version.

The only modification introduced in the code is the addition of a OpenMP pragma directive that distributes the computation of the for loop on each MPI process to different threads.

We can see that this directive affects only to the computation step of the algorithm. Once the for loop finishes in all the threads, only the master thread on each MPI process executes the collective communication operation.

2 Sparse matrix-vector product (VI)

Experimental environment

- *ra:* cluster with 34 dual-SMP nodes
- Node:
  - Two Intel Xeon Processors. 2.4 GHz
  - 1GByte RAM, 512 KByte L2 cache
- Myrinet network (2Gbps)
- Software:
  - Intel Compilers v.7.1. OpenMP
  - MPI v.1.2.5. dev: `ch_gm`

Notes

Now I will show the experimental results that we have obtained with different versions of the sparse matrix-vector product.

We performed the first experiments on a cluster of personal computers called ra.

This cluster contains 34 dual nodes.

Each node contains 2 Intel Xeon Processors with 1 gygabyte of memory.

All the nodes are connected through a Myrinet network.

2 Sparse matrix-vector product (VII)

Experimental tests

- *Uniprocessor:* One MPI process per node
- *Hybrid OpenMP+MPI:* One MPI process per node with two OpenMP threads: one per processor
- *Pure MPI:* Two MPI processes per node $\equiv$ one per processor

Notes

Our experimental tests compare three different versions of the algorithm.

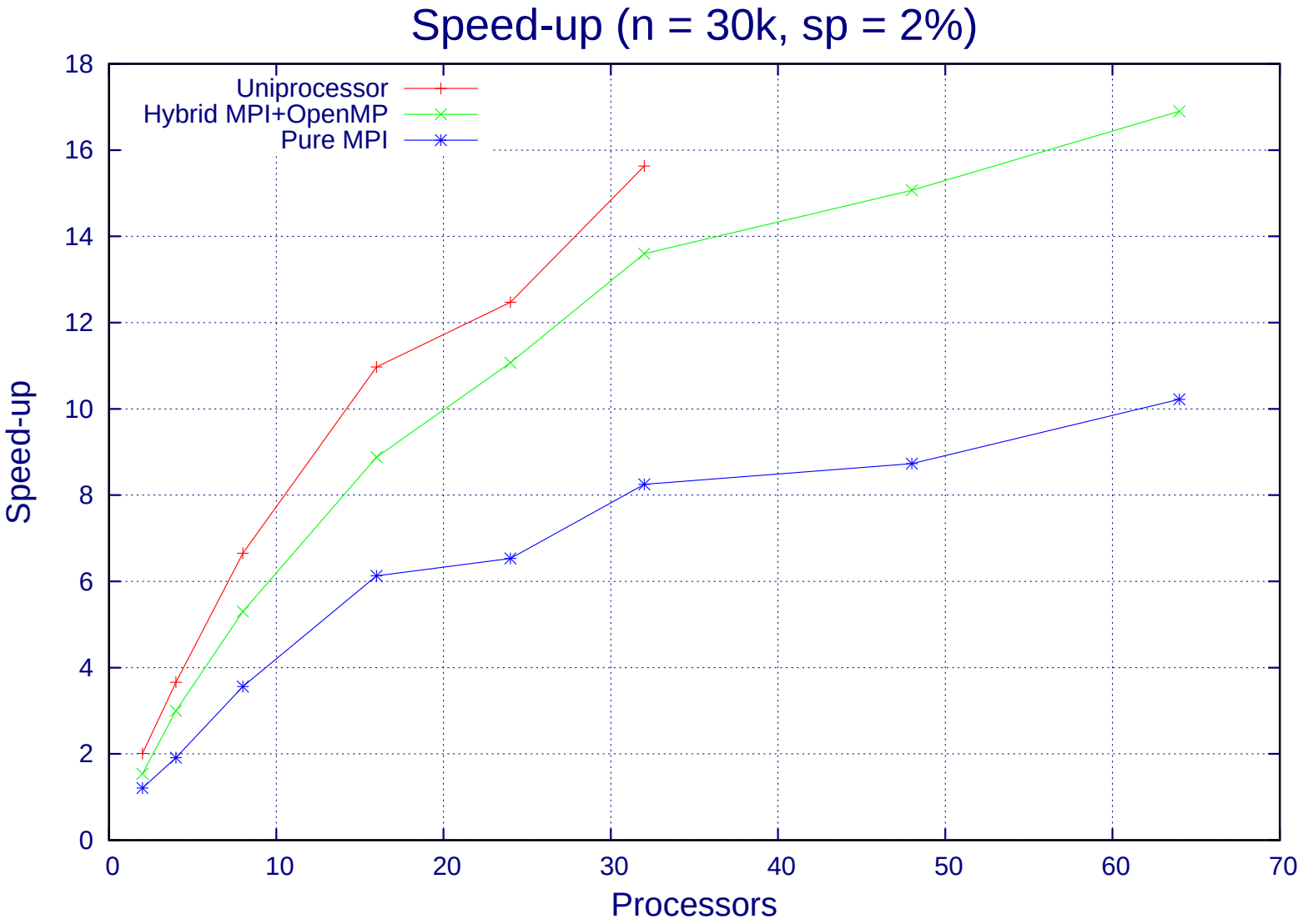
A uniprocessor version that sees the cluster as a cluster of uniprocessors and executes one MPI process on each node.

A hybrid version that executes a MPI process per node that runs two OpenMP threads, one per processor.

Finally, we test a Pure MPI version that executes 2 MPI processes per node, one per processor. This processes combine the communication through the network between the nodes and the communication using the shared memory within each node.

2 Sparse matrix-vector product (VIII)

Experimental Results in *ra*



Notes

In this figure we can see the speed-ups obtained with the three previous versions of the parallel product. We are performing a product with a sparse matrix with thirty thousand rows and an sparsity of 2 percent.

The first thing that we can see is that the speedups are far from the maximum. This is a logical result if we have into account that we are dealing with sparse matrices with a few nonzero elements. In this case, the cost of the communications has a big effect on the performance of the parallel algorithm.

The second thing that we can see is that, given a number of processors, the best results are obtained with the uniprocessor version of the algorithm. This also seems a logical result because in the other two versions of the algorithm different processes or threads are sharing the resources of each node, for example, the memory or the network bandwidth.

Notes

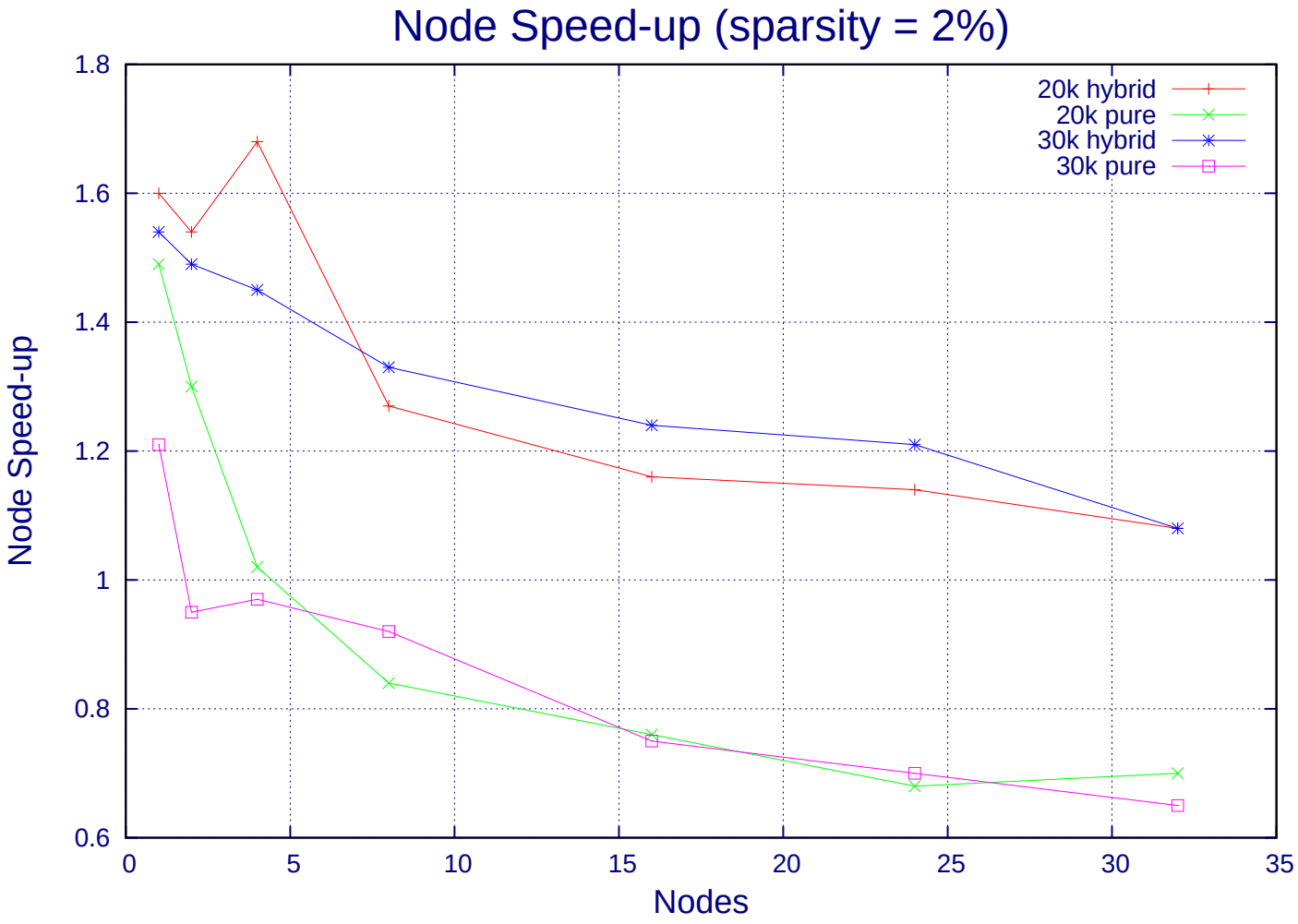
Notes

A third aspect to point out is that the hybrid version improves the results of the uniprocessor version when we increase the number of processors and exploit the two processors on each node. However, in this application and with this machine the improvement obtained with this version is very small.

Finally, we can see that the Pure MPI version exploiting the two processors on each node offers always worse results than the hybrid version and it never improves the results obtained with the uniprocessor version, even if we use twice as much processors.

2 Sparse matrix-vector product (IX)

Node Speedup: Time with 1 processors per node / Time with 2 processor per node.



Notes

In order to evaluate how the hybrid and pure mpi algorithms use the two processors on each node we have used the node speedup. This parameter is computed as the result of dividing the time spent by the algorithm that uses one processor per node and the algorithm using the two processors of each node.Theoretically the maximum value of this parameter is two if we are perfectly exploiting the two processors on each node.

In this slide we can see the node speedup of the algorithms with matrices of sizes 20 thousand and 30 thousand.

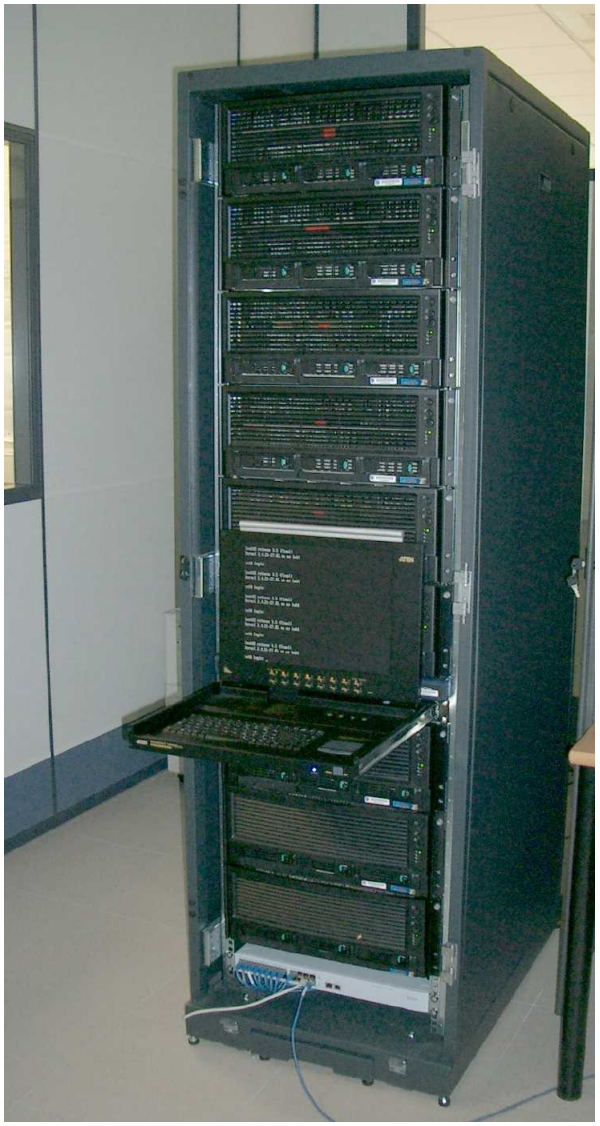
We can see that the node speedups obtained with the hybrid and with the Pure MPI versions are not very good and they tend decrease as we increase the number of nodes.

Besides, the Pure MPI version only obtains node speedups greater than one with few nodes. That means that with more than four processors it is slower than the uniprocessor version using the same number of nodes and two times more processors.

2 Sparse matrix-vector product (X)

Experimental environment

- *cat:* 4-way SMP cluster with 9 nodes.
- Node:
  - 4 *Itanium-2* processors. 1.5 Ghz.
  - 4GByte RAM, 4 MByte L3 cache.
- Infiniband network
- Software:
  - Intel Compilers v.8.1. OpenMP
  - MPI v. 1.2.5. dev: `vapi`



Notes

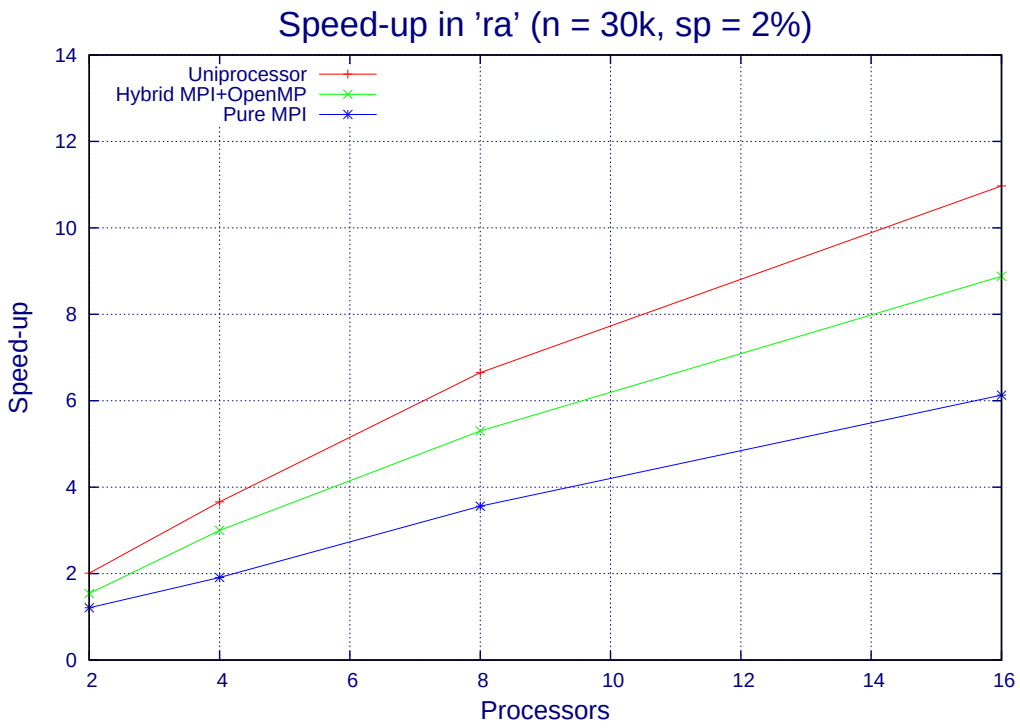
We were quite surprised by the bad behaviour of the hybrid and pure MPI versions of the algorithm with this cluster and we decided to test it in another cluster of personal computers called cat.

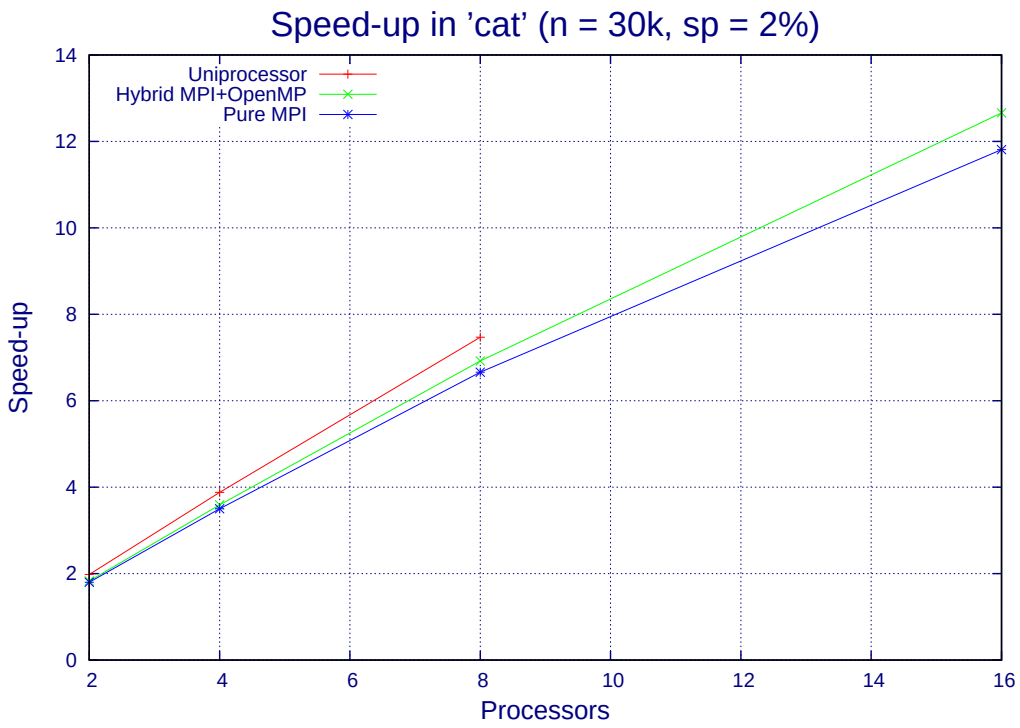
This is a cluster with 9 nodes.

Each node contains 4 itanum-2 processors with 4 Gigabytes of memory.

The communication network is a infiniband network and we have used a version of the MPI library specific to this kind of network and the OpenMP included in the Intel C compiler.

2 Sparse matrix-vector product (XI)





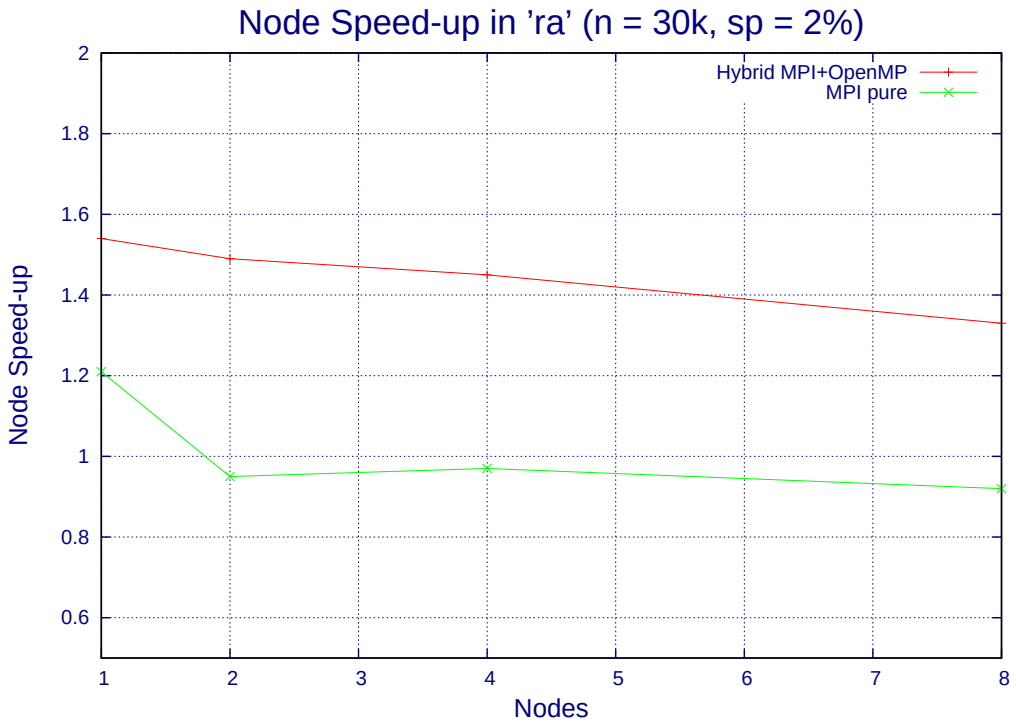
Notes

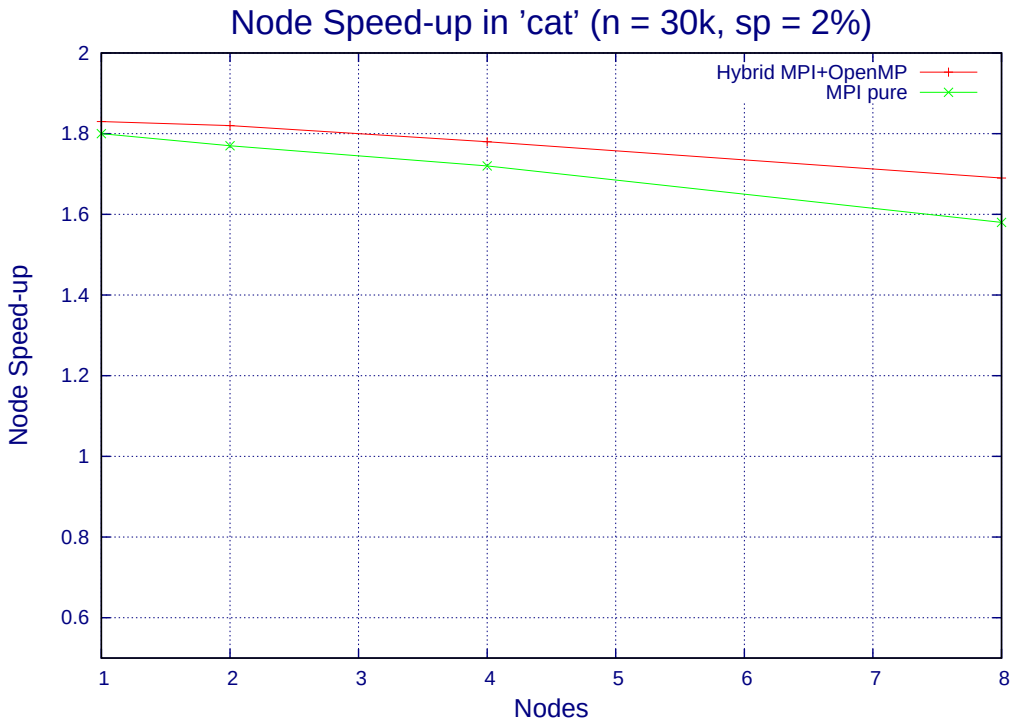
In this slide we compare the speedups obtained by the three versions of the parallel algorithm in the two clusters.

We can see that the speedups obtained with the second cluster are quite better than the speedups obtained by the first cluster.

However, the most important difference between the results in both clusters is the very different behavior of the algorithms using two processors per node. In the second cluster, the hybrid and pure mpi algorithms obtain speedups very similar to the uniprocessor algorithm with few processors, and both algorithms clearly improve the results obtained with the uniprocessor version when we increase the number of processors by using two processors per node.

2 Sparse matrix-vector product (XII)





Notes

The much better behaviour of the algorithms in the second cluster can be better seen if we compare the node speedups obtained in both clusters.

We can see that in the second cluster the node speedups of the hybrid and pure mpi versions of the product are quite close to 2 and they are only slightly reduced when we increase the number of nodes.

Therefore, we can conclude that the behaviour of the parallel versions of the product clearly depends on the machine, and mainly on the size of the memory in the nodes and on the behaviour of the network.

3 Dynamic programming

Example: Single Resource Allocation Problem

Allocate M units of indivisible resource to N tasks maximizing the effectiveness:

$$\max\ z = \sum_{j=1}^{N} f_j(x_j) \quad \text{subject to} \sum_{j=1}^{N} x_j = M,$$

$G[i][x]$ optimal benefit considering the first i tasks and x units of resource:

Dynamic programming recurrence equations:

$$\begin{aligned} G[i][x] &= \max\{G[i-1][x-j] + f_i(j) : 0 < j \leq x\}, \quad i = 2,\ldots,N, \\ G[1][x] &= f_1(x), \quad 0 < x \leq M, \quad \text{and } G[i][x] = 0, \quad i = 1,\ldots,N; \quad x = 0. \end{aligned}$$

$G[N][M]$: total income of the optimization problem.

Notes

In order to test if the behavior of the hybrid versions of the algorithm also depend on the application, we have implemented the hybrid solution of a different problem: The dynamic programming problem.

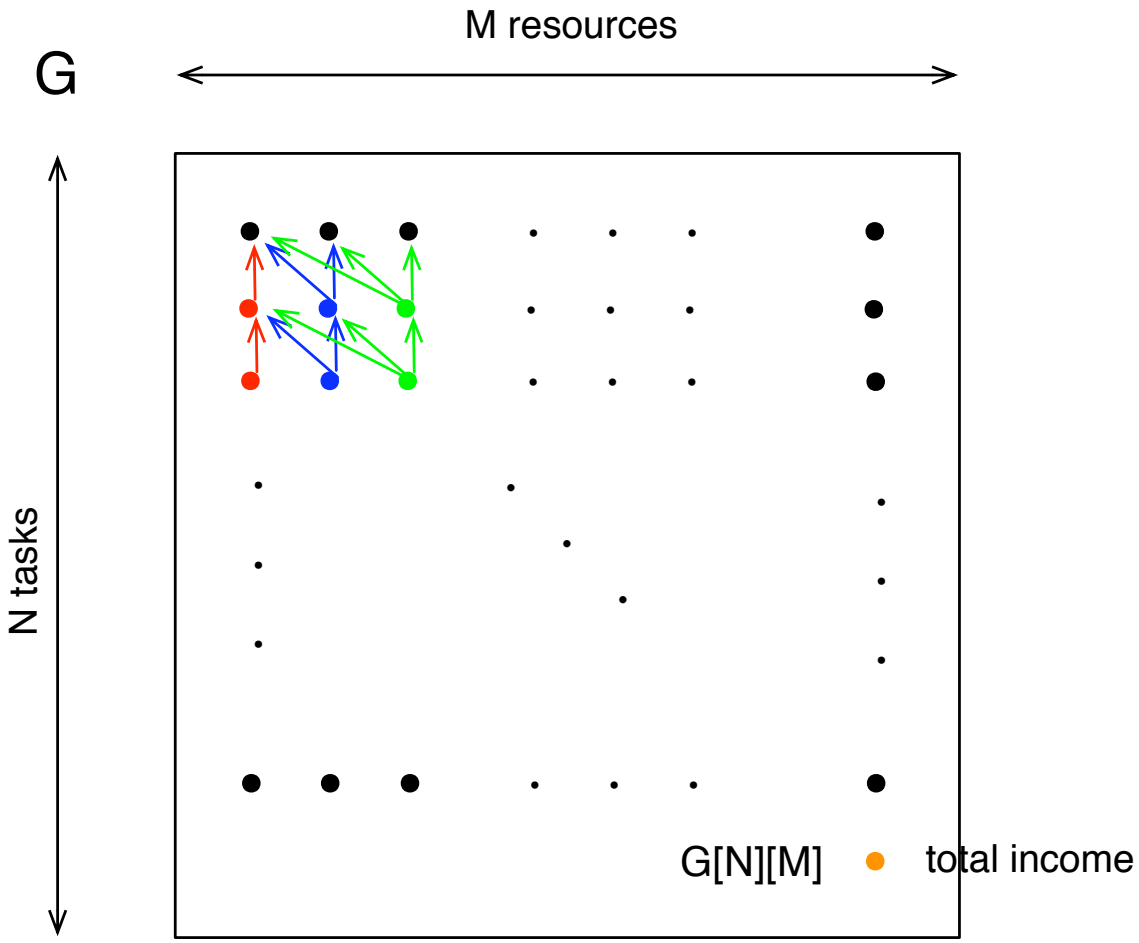
This is a very important problem solving technique that arises in multiple applications, like for example the single resource allocation problem.

In this problem we have to allocate M units of indivisible resource to N task maximizing the effectiveness.

To solve this problem we compute all the elements of a matrix G with M rows and N columns. The element i, x of the matrix represents the optimal benefit obtained taking into account the first i tasks and x units of resource.

Once computed the different values of the matrix, the value in the last position will give us the total income of the optimization problem.

Data dependences



Notes

The following slides will show the dependence of the different elements of the matrix.

The different rows of the matrix are computed sequentially.

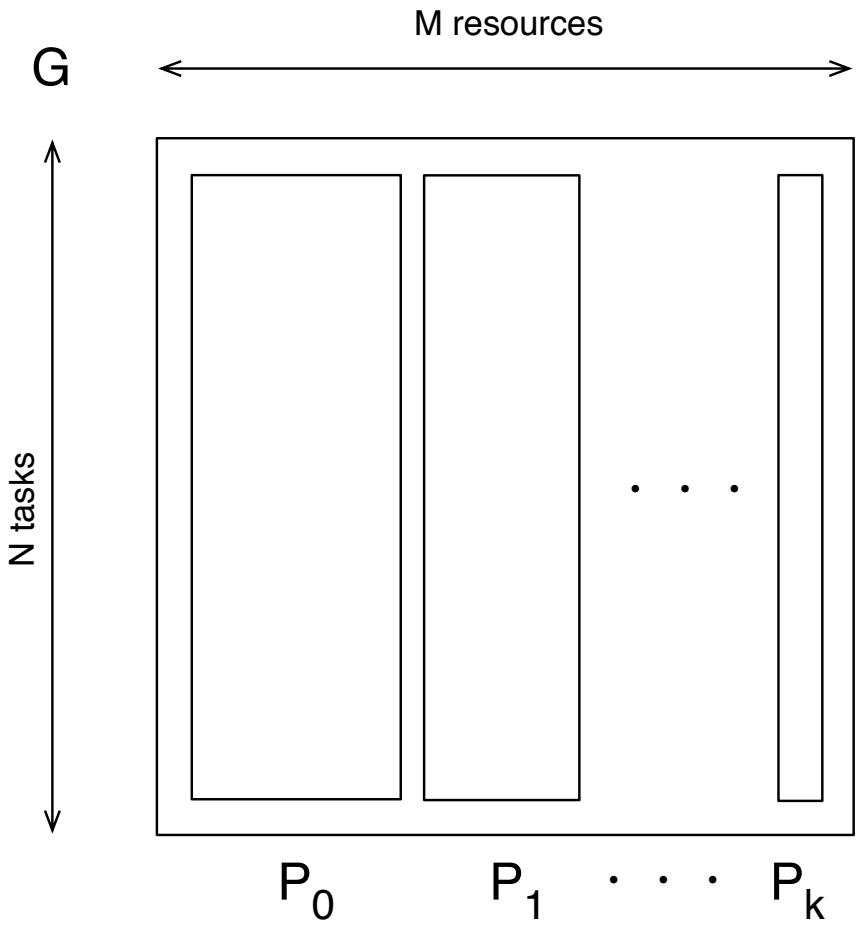
First, all the elements of the first row can be computed in parallel.

Then, we compute the first element of the second row, that depends on the first element of the previous row.

Next, we compute the second element of the second row, that depends on the first two elements of the previous row. Similarly, the third element of the second row depends on the first three elements of the first row, and so on.

We apply the same process to the different rows of the matrix until we have computed all its elements. Then the last element of the matrix gives us the optimal solution of the problem.

Data distribution: Decreasing blocks of columns



Notes

We have seen that the different rows of the matrix have to be computed sequentially, but once computed each row we can compute in parallel all the elements of the next row.

We have also seen that each element of each row depends on a different number of elements of the previous row.

Therefore, in order to balance the computations on each process we distribute the elements of the matrix by blocks of columns with decreasing size, as we can see in the slide.

Pure MPI code

```
for (i = 0; i <= N_TASKS; i++) {
  for (x = displs[myid]; x < displs[myid + 1]; x++) {
      G[i][x] = (*f)(i, 0);
      for (j = 0; j <= x; j++) {
          fix = G[i - 1][x - j] + (*f)(i, j);
          if (G[i][x] > fix)
              G[i][x] = fix;
      }
  }
  MPI_Allgatherv(&G[i][displs[myid]], ... );
}
```

Notes

In this slide we can see the pure mpi version of the code of the parallel solution of the dynamic programming problem. This code is executed by each MPI process.

The external loop is associated with the sequential computation of the successive rows of the matrix.

On each iteration of the loop we can see the two main steps of the algorithm.

During the first step all the process compute their columns and during the second step, the algorithm performs an allgather communication so that all the processes have all the elements of the row.

The indexes of the columns that have to be computed by each process are stored on a vector and used in the x for loop.

Hybrid MPI + OpenMP code

```
#pragma omp parallel private(...)
{
for (i = 0; i <= N_TASKS; i++) {
    for (x = th_displs[th_id]; x < th_displs[th_id + 1]; x++) {
          // idem than in the Pure MPI version
    }
    #pragma omp barrier
    #pragma omp master
      MPI_Allgatherv(&G[i][th_displs[th_id]], ... );
 } \\end of i loop
 }
```

Notes

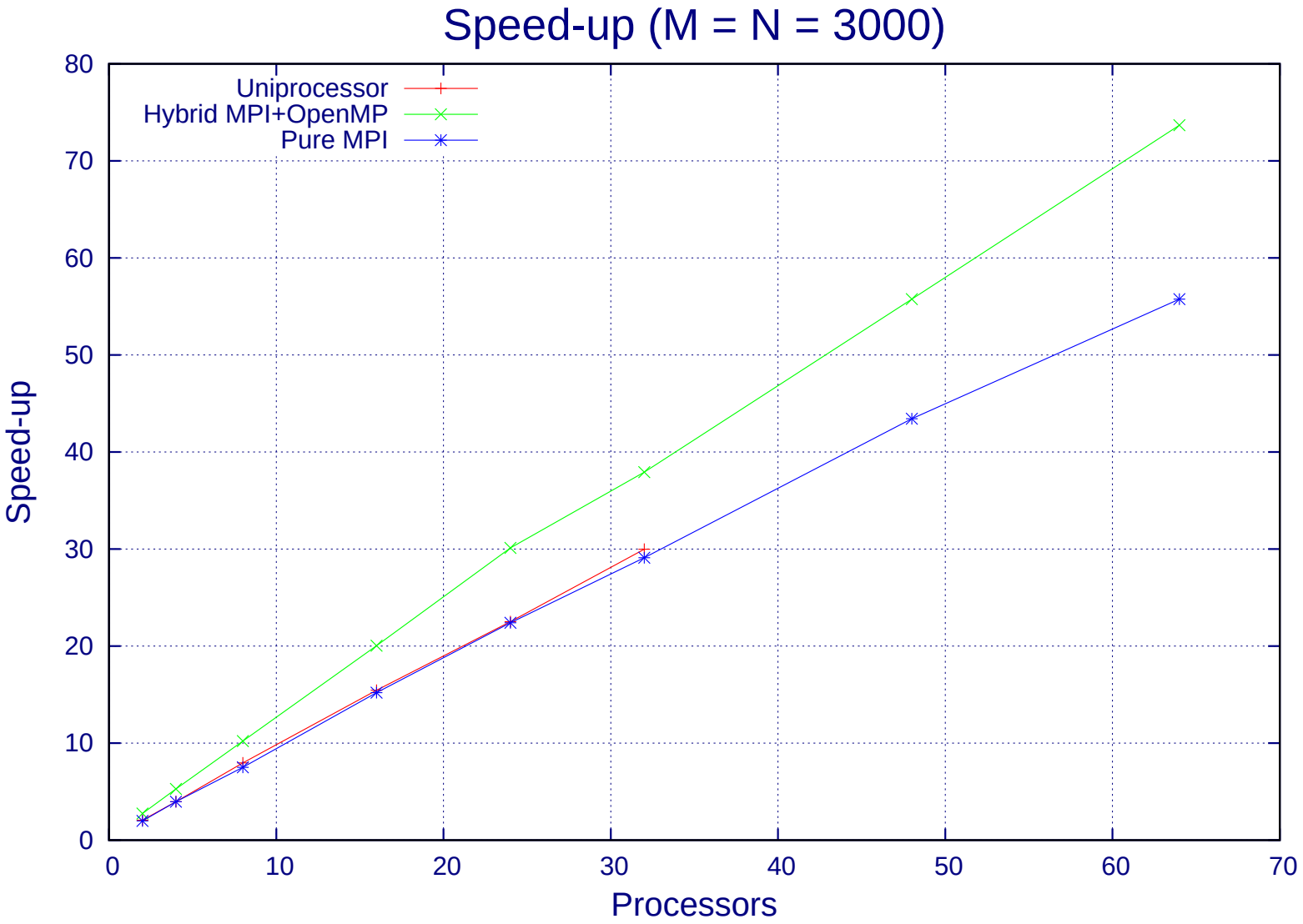
The hybrid version of the code applies the same kind of parallelism at two levels.

On each MPI process the computation of the different columns is distributed among several threads by using a OpenMP parallel directive.

On each iteration of the external loop all the threads are synchronized after finishing their computations by means of a barrier directive.

Finally, the master directive is used so that only the master thread on each MPI process participates on the collective communication step.

Experimental Results in *ra*



Notes

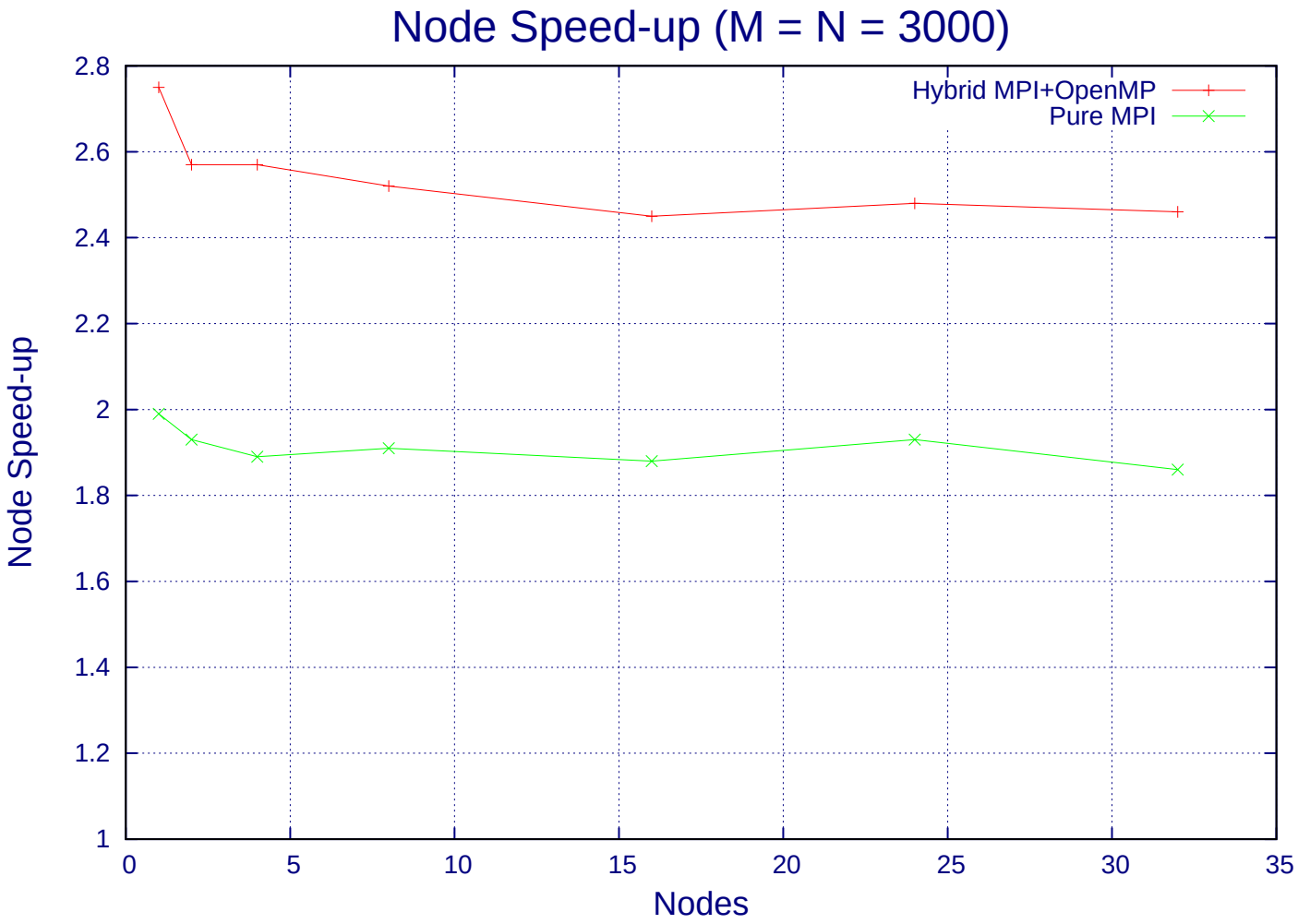
We have tested the behaviour of the same three versions of the algorithm than in the case of the sparse product.

The slide shows the results obtained when solving the problem with three thousand units of resource and three thousand tasks.

In this case the results on the first cluster of dual nodes are very good. We obtain speedups very close to the maximum even using all the processors of the cluster.

In some cases we obtain even superspeedups.

3 Dynamic programming (VII)



Notes

We can also see that the node speedups of the hybrid and the pure mpi algorithms are very good. And in this application the hybrid version of the algorithm obtains even node speedups larger than 2.

4 Conclusions

- Hybrid programming seems a natural, efficient and portable approach to program hybrid architectures.
- Performance of hybrid vs. pure MPI approaches depends on multiple factors at different levels:
  - The application and its parallelization scheme
  - The Operating System management of processes, threads, memory access, etc.
  - Architecture:
    - Memory size, hierarchy and performance
    - Network latency and bandwidth
    - Communication vs. computation cost
- There is much work to do related with hybrid programming.

Notes

This final slide shows the main conclusions of our research. First, the hybrid programming model seems a natural, efficient and portable approach to programming hybrid architectures.

The performance of the hybrid and the pure mpi approaches depend on multiple factors at different levels. We can point out for example the following factors:

* The application and its parallelization scheme.

* How the Operating system manages the processes, threads, memory accesses, etc.

* The results also depend on the architecture of the machine. For example, they can depend on the memory size and its performance, on the network latency and bandwidth and on the relation between the communication and the computation cost.

After implementing and testing two different problems on two different clusters we can conclude that there is still much work to do analyzing the influence of the different factors that can justify the behaviour of the hybrid algorithms.