Федеральное агентство связи

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

№ кода и наименование направления подготовки

**ОТЧЕТ**

по учебной практике

Выполнил:

студент гр. ИВ-621 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Дьяченко Д.В./

подпись

Руководитель практики от университета:

должность \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Родионов А.С./

ОЦЕНКА, подпись

Новосибирск 2018\_\_\_

1. Введение 3
2. Описание аппаратного и программного обеспечения
3. Краткое описание алгоритма
4. Результаты исследования
   1. При одном узле
   2. При двух узлах
   3. При четырех узлах
5. Выводы
6. Листинг
7. Введение

В рамках учебной практике были изучены основные функции стандарта MPI (Message Passing Interface). Был разработан алгоритм ленточного перемножения матриц с оптимизацией кэша и проведен эксперимент.

1. Описание аппаратного и программного обеспечения.

Исследование было произведено на вычислительном кластере Jet, укомплектованного 18 узлами.

Таблица 1 Конфигурация вычислительного узла

|  |  |
| --- | --- |
| Системная плата | Intel S5000VSA (Серверная платформа Intel SR2520SAF) |
| Процессор | 2 x Intel Xeon E5420 (2,5 GHz; Intel-64) |
| Оперативная память | 8 GB (4 x 2GB PC-5300) |
| Жесткий диск | SATAII 500GB (Seagate 500Gb Barracuda) |
| Сетевая карта | 2 x Intel Gigabit Ethernet (Integrated Intel PRO/1000 EB, 80003ES2LAN Gigabit Ethernet Controller) 1 x Intel PRO/1000 MT Server Adapter (PWLA8490MT, 82572EI Gigabit Ethernet Controller) |
| Корпус | Rack mount 2U |

Схема исследования:

N – количество элементов массива (каждый массив состоит из NxN элементов).

NumNodes – количество задействованных узлов.

NumCores – количество задействованных ядер процессора.

Таблица 2 Схема исследования

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NumNodes | NumCores | N | NumNodes | NumCores | N | NumNodes | NumCores | N |
| 1 | 1 | 1024 | 1 | 1 | 2048 | 1 | 1 | 4096 |
|  | 2 | 1024 |  | 2 | 2048 |  | 2 | 4096 |
|  | 4 | 1024 |  | 4 | 2048 |  | 4 | 4096 |
|  | 6 | 1024 |  | 6 | 2048 |  | 6 | 4096 |
|  | 8 | 1024 |  | 8 | 2048 |  | 8 | 4096 |
| 2 | 1 | 1024 | 2 | 1 | 2048 | 2 | 1 | 4096 |
|  | 2 | 1024 |  | 2 | 2048 |  | 2 | 4096 |
|  | 4 | 1024 |  | 4 | 2048 |  | 4 | 4096 |
|  | 6 | 1024 |  | 6 | 2048 |  | 6 | 4096 |
|  | 8 | 1024 |  | 8 | 2048 |  | 8 | 4096 |
| 4 | 1 | 1024 | 4 | 1 | 2048 | 4 | 1 | 4096 |
|  | 2 | 1024 |  | 2 | 2048 |  | 2 | 4096 |
|  | 4 | 1024 |  | 4 | 2048 |  | 4 | 4096 |
|  | 6 | 1024 |  | 6 | 2048 |  | 6 | 4096 |
|  | 8 | 1024 |  | 8 | 2048 |  | 8 | 4096 |
| 8 | 1 | 1024 | 8 | 1 | 2048 | 8 | 1 | 4096 |
|  | 2 | 1024 |  | 2 | 2048 |  | 2 | 4096 |
|  | 4 | 1024 |  | 4 | 2048 |  | 4 | 4096 |
|  | 6 | 1024 |  | 6 | 2048 |  | 6 | 4096 |
|  | 8 | 1024 |  | 8 | 2048 |  | 8 | 4096 |

Программа была написана на языке программирования Си, с использованием компилятора mpicc. Время замерялось при помощи функции wtime из библиотеки sys/time.h.

**double** wtime()

{

**struct** timeval t;

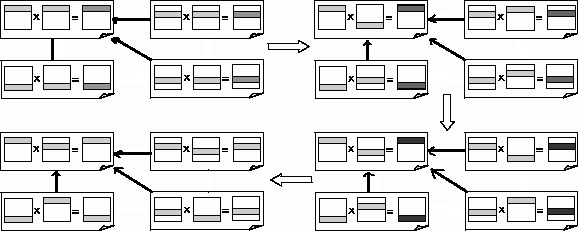
gettimeofday(&t, NULL);

return (**double**)t.tv\_sec + (**double**)t.tv\_usec \* 1E-6;

}

1. Краткое описание алгоритма
2. Инициализация данных для обработки функцией init\_matrix (массивов A[N\*N], B[N\*N], C[N\*N]). Для эксперимента использовались статичные данные.
   1. **void** init\_matrix(**double** \*a, **double** \*b, **double** \*c, **int** n)
   2. {
   3. **int** i, j, k;
   4. for (i = 0; i < n; i++) {
   5. for (j = 0; j < n; j++) {
   6. for (k = 0; k < n; k++) {
   7. \*(a + i \* n + j) = 1.0;
   8. \*(b + i \* n + j) = 2.0;
   9. \*(c + i \* n + j) = 0.0;
   10. }
   11. }
   12. }
   13. }
3. Нахождение номера начала и конца строки, необходимого для обработки, основываясь на количестве процессов и номере процесса, выполняемого код в данный момент.
   1. **int** rank;
   2. **int** size;
   3. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
   4. MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);
   5. **int** st = (N / size) \* rank;
   6. **int** fn = ((rank + 1) == size) ? N : (st + (N / size));
4. Выполнение умножения матриц функцией dgemm\_parallel (цикл ikj).
   1. **void** dgemm\_parallel(**double** \*a, **double** \*b, **double** \*c, **int** n, **int** st, **int** fn)
   2. {
   3. **int** i, j, k;
   5. for (i = st; i < fn; i++) {
   6. for (k = 0; k < n; k++) {
   7. for (j = 0; j < n; j++) {
   8. \*(c + i \* n + j) += \*(a + i \* n + k) \* \*(b + k \* n + j);
   9. }
   10. }
   11. }
   12. }
5. Отправка подсчитанных строк массива в главный узел для объединения данных остальными процессами и принятие этим процессом данных.
   1. if (rank != 0) {
   2. MPI\_Send(&t, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);
   3. }
   4. if (rank == 0) {
   5. for (**int** i = 1; i < size; i++) {
   6. MPI\_Recv(&t\_all[i], 1, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);
   7. }
   8. }
6. Завершение, очистка памяти
   1. MPI\_Finalize();
   2. free(A);
   3. free(B);
   4. free(C);

Рисунок 1 Общая схема алгоритма перемножения матриц



1. Результаты исследования

N – количество элементов массива (каждый массив состоит из NxN элементов).

NumCores – количество задействованных ядер процессора.

Time – время выполнения

Speedup – коэффициент ускорения, рассчитываемый по формуле , где p – количество процессов.

* 1. При одном узле

Таблица 3 Таблица времени и ускорения на одном узле для N=1024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 1024 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 10.738666 | 5.371339 | 2.814382 | 1.848182 | 1.477680 |
| Speedup | 1 | 1.999253073 | 3.815639099 | 5.810394214 | 7.267247307 |

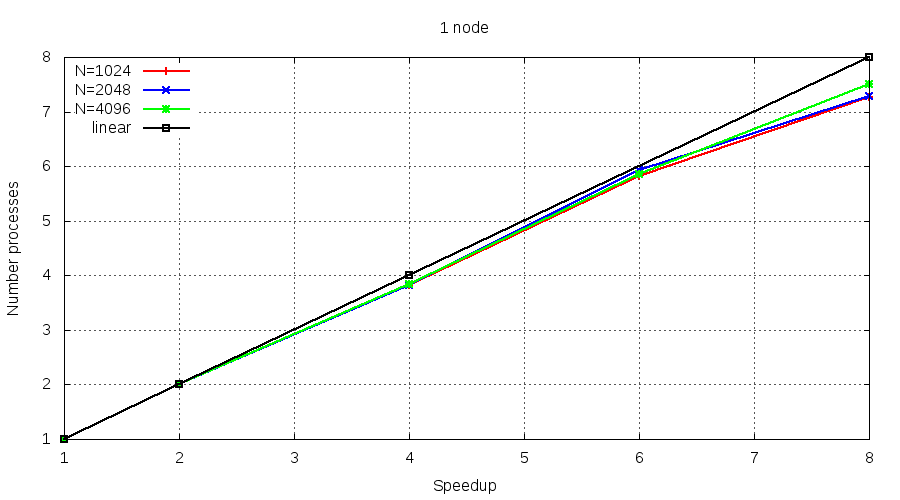
Таблица 4 Таблица времени и ускорения на одном узле для N=2048

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 2048 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 86.507901 | 43.158811 | 22.622286 | 14.599645 | 11.875154 |
| Speedup | 1 | 2.00440881 | 3.824012348 | 5.925342774 | 7.284781402 |

Таблица 5 Таблица времени и ускорения на одном узле для N=4096

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 4096 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 696.882661 | 347.676537 | 181.444636 | 118.917351 | 92.918897 |
| Speedup | 1 | 2.004399454 | 3.840745455 | 5.860226915 | 7.499902426 |

Рисунок 2 График зависимости коэффициента ускорения от числа процессоров



* 1. При двух узлах

Таблица 6 Таблица времени и ускорения на двух узлах для N=1024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 1024 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 10.738666 | 5.371339 | 2.814382 | 1.848182 | 1.477680 |
| Speedup | 1 | 1.999253073 | 3.815639099 | 5.810394214 | 7.267247307 |

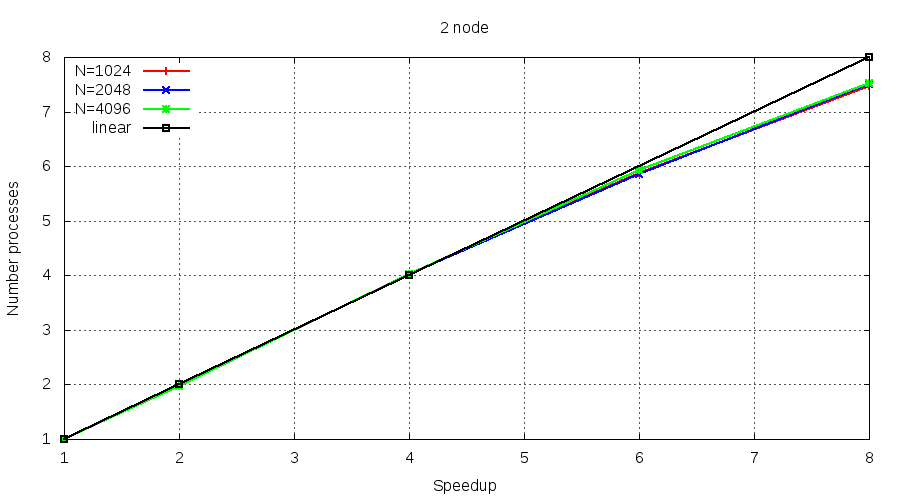
Таблица 7 Таблица времени и ускорения на двух узлах для N=2048

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 2048 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 86.507901 | 43.158811 | 22.622286 | 14.599645 | 11.875154 |
| Speedup | 1 | 2.00440881 | 3.824012348 | 5.925342774 | 7.284781402 |

Таблица 8 Таблица времени и ускорения на двух узлах для N=4096

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 4096 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 696.882661 | 347.676537 | 181.444636 | 118.917351 | 92.918897 |
| Speedup | 1 | 2.004399454 | 3.840745455 | 5.860226915 | 7.499902426 |

Рисунок 3 График зависимости коэффициента ускорения от числа процессоров



* 1. При четырех узлах

Таблица 9 Таблица времени и ускорения на двух узлах для N=1024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 1024 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 2.841816 | 1.442992 | 0.670140 | 0.607875 | 0.425006 |
| Speedup | 1 | 1.969391376 | 4.240630316 | 4.675000617 | 6.686531484 |

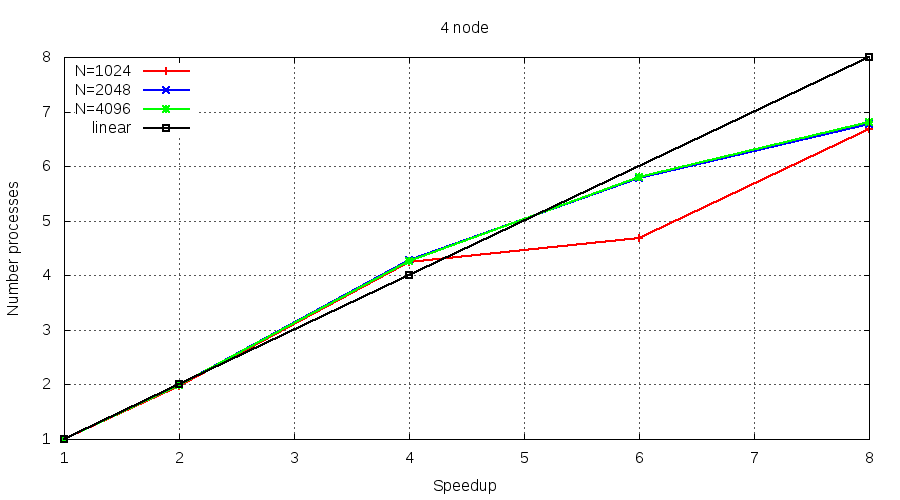
Таблица 10 Таблица времени и ускорения на двух узлах для N=2048

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 2048 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 22.981416 | 11.552278 | 5.374940 | 3.976240 | 3.389849 |
| Speedup | 1 | 1.98934063 | 4.275660007 | 5.779685331 | 6.779578736 |

Таблица 11 Таблица времени и ускорения на двух узлах для N=4096

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 4096 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 184.620054 | 92.616670 | 43.326093 | 31.847790 | 27.112465 |
| Speedup | 1 | 1.993378233 | 4.261174761 | 5.796950244 | 6.809416038 |

Рисунок 4 График зависимости коэффициента ускорения от числа процессоров



* 1. При восьми узлах

Таблица 12 Таблица времени и ускорения на восьми узлах для N=1024

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 1024 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 10.738666 | 5.371339 | 2.814382 | 1.848182 | 1.477680 |
| Speedup | 1 | 1.999253073 | 3.815639099 | 5.810394214 | 7.267247307 |

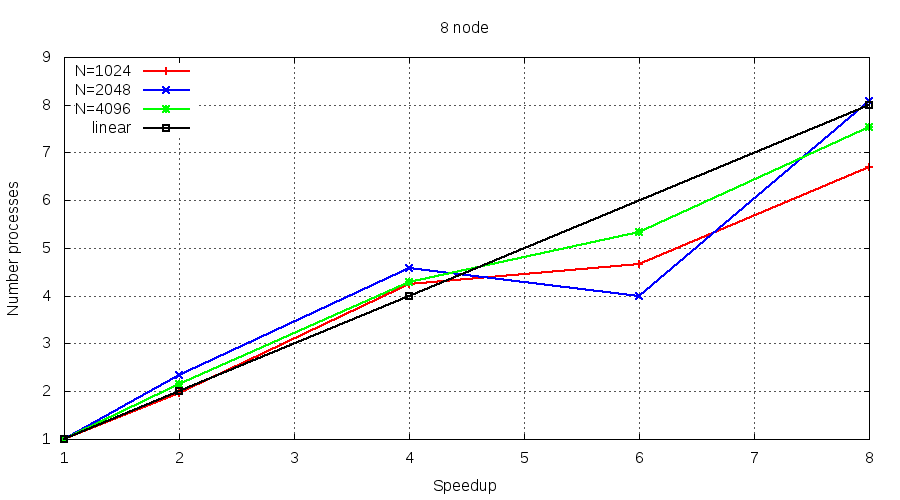
Таблица 13 Таблица времени и ускорения на восьми узлах для N=2048

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 2048 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 86.507901 | 43.158811 | 22.622286 | 14.599645 | 11.875154 |
| Speedup | 1 | 2.00440881 | 3.824012348 | 5.925342774 | 7.284781402 |

Таблица 14 Таблица времени и ускорения на восьми узлах для N=4096

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 4096 | | | | |
| NumCores | 1 | 2 | 4 | 6 | 8 |
| Time | 696.882661 | 347.676537 | 181.444636 | 118.917351 | 92.918897 |
| Speedup | 1 | 2.004399454 | 3.840745455 | 5.860226915 | 7.499902426 |

Рисунок 5 График зависимости коэффициента ускорения от числа процессоров



1. Выводы

Результаты показывают хорошую эффективность распараллеливания, близкую к линейной на количестве узлов 1 и 2, но на 4 и 8 с 4 активными ядрами графики показывают суперскалярное ускорение. Такая ситуация возникает при распределении 1:1, то есть один процесс на один узел, что дает увеличение объема, используемого одним процессом кэша. Кэши второго и третьего уровней инклюзивные, а в ситуации, когда используется лишь одно ядро, происходит меньше кэш-промахов, так как большую часть кэшей занимают данные активного ядра. Обратная ситуация при 4 и более активных процессах на количестве узлов 4 и 8. Дополнительные задержки появляются в результате затрат на создание нового процесса и увеличенном количестве пересылок данных между узлами по довольно медленному интерфейсу.

Данный алгоритм имеет зависимость от подсистемы памяти, поэтому, чтобы минимизировать задержки здесь применяется параллельная инициализация данных. Кэш процессоров «прогретый», то есть уже частично имеет данные необходимые для дальнейшей обработки. Так же умножение происходит типом строка-строка, а не строка-столбец, что дает эффективнее использовать кэш. А благодаря библиотеке MPI данный алгоритм имеет хорошую масштабируемость, которую можно проследить по таблицам и графикам.

Даже такая, довольно нетривиальная задача, показывает производительность, гибкость и масштабируемость данной библиотеки.

1. Листинг
2. #include <stdio.h>
3. #include <mpi.h>
4. #include <time.h>
5. #include <sys/time.h>
6. **enum** {
7. N = 1024, */\* 2048, 4096 \*/*
8. NREPS = 5
9. };
10. **double** A[N \* N], B[N \* N], C[N \* N];
11. **void** dgemm\_parallel(**double** \*a, **double** \*b, **double** \*c, **int** n, **int** st, **int** fn)
12. {
13. **int** i, j, k;
15. for (i = st; i < fn; i++) {
16. for (k = 0; k < n; k++) {
17. for (j = 0; j < n; j++) {
18. \*(c + i \* n + j) += \*(a + i \* n + k) \* \*(b + k \* n + j);
19. }
20. }
21. }
22. }
23. **void** init\_matrix(**double** \*a, **double** \*b, **double** \*c, **int** n)
24. {
25. **int** i, j, k;
26. for (i = 0; i < n; i++) {
27. for (j = 0; j < n; j++) {
28. for (k = 0; k < n; k++) {
29. \*(a + i \* n + j) = 1.0;
30. \*(b + i \* n + j) = 2.0;
31. \*(c + i \* n + j) = 0.0;
32. }
33. }
34. }
35. }
36. **void** printMatrix(**double** \*a, **int** n)
37. {
38. for (**int** i = 0; i < n; i++) {
39. for (**int** j = 0; j < n; j++) {
40. printf("%.2lf ", \*(a + i \* n + j));
41. }
42. printf("\n");
43. }
44. printf("\n");
45. }
46. **double** wtime()
47. {
48. **struct** timeval t;
49. gettimeofday(&t, NULL);
50. return (**double**)t.tv\_sec + (**double**)t.tv\_usec \* 1E-6;
51. }
52. **int** main(**int** argc, **char** \*argv[])
53. {
54. MPI\_Init(&argc, &argv);
55. **double** t;
56. **int** rank;
57. **int** size;
58. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
59. MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);
60. **double** t\_all[size];
61. init\_matrix(A, B, C, N);
62. **int** st = (N / size) \* rank;
63. **int** fn = ((rank + 1) == size) ? N : (st + (N / size));
64. t -= wtime();
65. for (**int** i = 0; i < NREPS; i++) {
66. dgemm\_parallel(A, B, C, N, st, fn);
67. }
68. t += wtime();
69. t /= NREPS;
70. MPI\_Barrier(MPI\_COMM\_WORLD);
71. if (rank != 0) {
72. MPI\_Send(C + (st \* N), (fn - st) \* N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);
73. }
74. MPI\_Status status;
75. if (rank == 0) {
76. for (**int** i = 1; i < size; i++) {
77. **int** st\_loc = (N / size) \* i;
78. **int** fn\_loc = ((i + 1) == size) ? N : (st\_loc + (N / size));
79. MPI\_Recv(C + (st\_loc \* N), (fn\_loc - st\_loc) \* N, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);
80. }
81. }
82. MPI\_Barrier(MPI\_COMM\_WORLD);
83. if (rank != 0) {
84. MPI\_Send(&t, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);
85. }
86. if (rank == 0) {
87. for (**int** i = 1; i < size; i++) {
88. MPI\_Recv(&t\_all[i], 1, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);
89. }
90. }
91. if (rank == 0) {
92. t\_all[0] = t;
93. }
94. **double** max = -1;
95. for (**int** i = 0; i < size; i++) {
96. if (t\_all[i] > max) {
97. max = t\_all[i];
98. }
99. }
100. if (rank == 0) {
101. printf("time = %.6f\n", max);
102. }
103. MPI\_Finalize();
105. return 0;
106. }